



University
of Glasgow

Perry, Thomas Paul (2013) Software tools for the rapid development of signal processing and communications systems on configurable platforms. EngD thesis

<http://theses.gla.ac.uk/4301/>

Copyright and moral rights for this thesis are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Software tools for the rapid development of signal
processing and communications systems on
configurable platforms

Thomas Paul Perry, M.A.

Submitted in fulfilment of the requirements for the EngD degree

Institute for System Level Integration

2013

Abstract

Programmers and engineers in the domains of high performance computing (HPC) and electronic system design have a shared goal: to define a structure for coordination and communication between nodes in a highly parallel network of processing tasks. Practitioners in both of these fields have recently encountered additional constraints that motivate the use of multiple types of processing device in a *hybrid* or *heterogeneous platform*, but constructing a working “program” to be executed on such an architecture is very time-consuming with current domain-specific design methodologies.

In the field of HPC, research has proposed solutions involving the use of alternative computational devices such as FPGAs (field-programmable gate arrays), since these devices can exhibit much greater performance per unit of power consumption. The appeal of integrating these devices into traditional microprocessor-based systems is mitigated, however, by the greater difficulty in constructing a system for the resulting hybrid platform.

In the field of electronic system design, a similar problem of integration exists. Many of the highly parallel FPGA-based systems that Xilinx and its customers produce for applications such as telecommunications and video processing require the additional use of one or more microprocessors, but coordinating the interactions between existing FPGA cores and software running on the microprocessors is difficult.

The aim of my project is to improve the design flow for hybrid systems by proposing, firstly, an abstract representation of these systems and their components which captures in metadata their different models of computation and communication; secondly, novel design checking, exploration and optimisation techniques based around this metadata; and finally, a novel design methodology in which component and system metadata is

used to generate software simulation models.

The effectiveness of this approach will be evaluated through the implementation of two physical-layer telecommunications system models that meet the requirements of the 3GPP “LTE” standard, which is commercially relevant to Xilinx and many other organisations.

Acknowledgements

There are a number of people who have offered me a great deal of assistance in the course of my EngD project, without which the production of this thesis would not have been possible.

Firstly, I would like to thank my EngD industrial supervisor, Richard Walke, who worked tirelessly to provide support and direction to my research. I am also grateful to Khaled Benkrid and Mark Parsons for their additional supervision and feedback, and to Rob Payne for his advice relating to the LTE application.

I would also like to thank the various members of Xilinx Edinburgh, and specifically the Wireless and DSP group, for their guidance and feedback on my work and their friendship. In particular, I am grateful to Graham Johnston for his assistance with the LTE Downlink Transmit model and to David Andrews for his help with the XMODEL framework. I'm also grateful to Bill Wilkie, Chris Dick and other members of Xilinx management for giving their permission for the project to take place, and to the senior technical staff in Xilinx San Jose, including Ralph Wittig, Gordon Brebner and Nabeel Shirazi amongst others, for their feedback on my project and for their guidance in targeting my work to the needs of Xilinx. Finally, Stefan Petko, Dave Fraser and Ben Jones deserve my thanks for improving my pool skills.

Other acknowledgements include Matt Lewis, for introducing me to SMT solvers, my examiners for suggesting a number of insightful improvements to my thesis, and to Siân Williams for her exceptionally responsive administration of the EngD programme. I would also like to thank the Engineering and Physical Sciences Research Council and Scottish Enterprise for funding my studies, and the Royal Commission for the Exhibition

of 1851 for their very generous Industrial Fellowship award.

Finally, on a personal note, I would like to thank my parents for always encouraging me, and I'd like to thank Emily for her love and support, which are more meaningful to me than I can express in words.

Declaration of Originality

I declare that, except where explicit reference is made to the contribution of others, that this thesis is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Contents

Abstract	2
Acknowledgements	4
Declaration of Originality	6
List of Figures	11
List of Tables	13
List of Code Listings	14
List of Abbreviations	15
1 Introduction	20
1.1 Configurable logic	21
1.2 Heterogeneous platforms	23
1.3 Programming abstractions	24
1.4 Goals	27
1.5 Contributions	27
1.6 Thesis outline	28
2 Challenges in designing signal processing systems	29
2.1 Xilinx LTE baseband systems	29
2.1.1 Coding, modulation and MIMO	31
2.1.2 Channel multiplexing and multiple access	33
2.2 Challenges in LTE system design	35

2.2.1	Automatic integration of IP cores	35
2.2.2	Design space exploration and optimisation	37
2.2.3	Software modelling	39
2.2.4	Heterogeneous processing	42
2.3	Summary	42
3	Background	44
3.1	Platform-based design and IP reuse	44
3.1.1	Interface specification in IP-XACT	46
3.2	Designing new components	50
3.2.1	High-level synthesis	52
3.2.2	Metamodelling	53
3.3	Composition of components	55
3.4	Models of computation	57
3.4.1	Process networks and actor-oriented design	58
3.5	Platform mapping	61
3.6	Summary of existing design tools	62
3.7	Conclusion	65
4	Architecture	66
4.1	Idealised design process	66
4.2	Tool flow overview	68
4.2.1	Intermediate representation	69
4.2.2	High-level inputs	72
4.2.3	Optimisations	73
4.2.4	Code generation	73
4.3	Implementation aspects	74
4.4	Conclusion	76
5	A metamodel for Xilinx IP cores and systems and its representation in extended IP-XACT	79
5.1	Requirements	80

5.2	Data type specification	85
5.2.1	Basic type descriptions	87
5.2.2	Hierarchical composition of types	89
5.2.3	Data type encoding	90
5.2.4	Naming and reference	91
5.2.5	Parameterisation and dependencies	93
5.2.6	Full examples	96
5.3	Component behaviour specification	99
5.3.1	Rate relationships	99
5.3.2	Dynamic data dependencies	101
5.3.3	Timing constraints	102
5.3.4	Blocking	105
5.3.5	Summary	109
5.4	Discussion	110
5.5	Conclusion	111
6	Tool-assisted design of multidimensional streaming systems	113
6.1	Determining buffering requirements	114
6.2	Automatic buffer instantiation	116
6.2.1	Chaining SID cores	117
6.2.2	Custom implementation	118
6.3	Determining repetition lists using pairwise propagation	120
6.4	Determining repetition lists using Synchronous Dataflow	122
6.4.1	Determining repetition sets automatically	124
6.4.2	Determining buffer-minimising repetition lists	125
6.4.3	Solver output	130
6.5	Inferring efficient data ordering to assist in component implementation . .	131
6.6	Eliminating redundant calculations	133
6.6.1	Generalisation to arbitrary generation and reduction functions . .	136
6.7	Implementation considerations	138
6.8	Discussion	138

6.9	Conclusion	139
7	A software model generation framework based on extended IP-XACT	140
7.1	Simple leaf-level components	141
7.1.1	Input language	141
7.1.2	XMODEL code generation	143
7.2	Data type input and code generation	146
7.2.1	Input language	146
7.2.2	XMODEL code generation	149
7.3	Integrating bit-accurate core simulation models	151
7.3.1	Actions	152
7.3.2	Parameters	154
7.3.3	Fully automatic integration of software models	155
7.4	Hierarchical components and scheduling	156
7.4.1	Input language	157
7.4.2	XMODEL code generation	158
7.4.3	Enforcing correct code generation order	161
7.5	Action guards	162
7.6	Remaining components	164
7.6.1	Subframe memory controller	164
7.6.2	OFDM	164
7.7	Integration into Vivado tool suite	165
7.8	Test methodology	166
7.9	Results	167
7.10	Discussion	168
7.11	Conclusions	168
8	Conclusion	169
8.1	Limitations and future work	170
	Bibliography	173
A	Publications	185

List of Figures

1.1	Visualisation of the available parallelism at various abstraction layers for two different architectures.	26
2.1	The Xilinx LTE downlink transmit and uplink receive systems.	31
2.2	The LTE uplink resource grid.	34
2.3	A data type interoperability issue in the LTE uplink receive system, caused by different orderings of array dimensions.	37
2.4	Solution to the interoperability issue through the addition of a reorder buffer.	38
2.5	Resource demapper output type is dependent on types of neighbours.	38
2.6	Example XMODEL component definitions.	40
3.1	High-level synthesis and metamodelling.	51
4.1	An idealised design process that addresses the issues raised in Chapter 2. Some of the new characteristics introduced in each stage are circled.	67
4.2	Tool flow overview.	69
4.3	Tool flow architecture, showing contributions.	77
5.1	FIR Compiler interfaces and data formats.	80
5.2	DUC/DDC Compiler data format in two modes: no TDM and 2 antennas; TDM and 4 antennas.	83
5.3	Metadata representation of complex array strides.	92
6.1	Propagation of dimensions.	116
6.2	Addition of Uplink Channel Decoder block.	121

6.3	Alternative propagation.	121
6.4	Efficient RDL calculated for Resource Demapper.	133
6.5	Calculation of IDLs from IDSs instead of RDL from RDS for the Resource Demapper leads to lower buffering requirements.	134
6.6	RD/CE/MIMO/IDFT/CD system with symbol replicator.	135
6.7	The (sym) dimension removed from the Channel Estimator.	136
6.8	GUI mock-up of a mechanism for specifying arbitrary data generation and reduction functions.	137
6.9	GUI mock-up showing interface dimension mapping.	138
7.1	Data type metadata and code generation flow (the significance of the colours is as shown in Figure 4.3).	147
7.2	Component metadata and code generation flow (the significance of the colours is as shown in Figure 4.3).	153
7.3	SCH encoder and modulation chain.	157

List of Tables

2.1	Interface characterisation of Xilinx DSP cores	36
3.1	Examples of model transformations.	55
3.2	Feature matrix for a selection of existing design automation tools.	64
5.1	Leaf-level data types in Xilinx DSP cores	82
5.2	Data interface array dimensionality in Xilinx DSP cores	84
5.3	A selection of data interface formats used in Xilinx video processing cores	86
5.4	Supported data format codes in a selection of Xilinx video cores	86
5.5	Dynamic interface behaviour of Xilinx DSP cores	109
6.1	High-level synthesis results for simple reorder buffer.	119
6.2	High-level synthesis results for optimised reorder buffer.	119
6.3	RDL results for MIMO/IDFT system.	130
6.4	RDL results for MIMO/IDFT/Channel Decoder system.	131
6.5	RDL results for RD/CE/MIMO/IDFT/CD system.	132
6.6	RDL/IDL results for RD/CE/MIMO/IDFT/CD system.	133
6.7	RDL results for RD/CE/MIMO/IDFT/CD system with symbol replicator.	135

List of Code Listings

3.1	IP-XACT component description example.	47
3.2	Representation of an optional bus interface in IP-XACT using vendor extensions.	48
3.3	IP-XACT design description example.	49
5.1	Metadata representation of a simple data type.	87
5.2	Abstract metadata representation of optional port.	94
5.3	Metadata representation of parameterised array dimension size and presence.	95
5.4	Metadata description of the CONFIG control packet received by FIR cores.	97
5.5	DUC/DDC Compiler data format expressed in XML metadata.	98
6.1	Reorder buffer described as C code for input to Vivado HLS.	118
6.2	Python session demonstrating symbolic determination of an SDF rate vector and an RDS.	125
6.3	Python function used to generate Z3 cost constraints.	129
6.4	Output from solver when applied to the MIMO/IDFT system.	130
6.5	Output from solver when applied to the MIMO/IDFT/Channel Decoder system.	131
6.6	Modified data copying statements for input to Vivado HLS.	136
6.7	C code for custom replicate-and-reorder buffer to be input to Vivado HLS.	137
7.1	NL representation of SCH modulation hierarchical block.	158

List of Abbreviations

AMC	Adaptive Modulation and Coding
AMD	Advanced Micro Devices
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ARM	ARM Ltd. (originally Advanced RISC Machines)
ARQ	Automatic Repeat Request
ASIC	Application-Specific Integrated Circuit
ASN.1	Abstract Syntax Notation One
AXI	ARM “Advanced Extensible Interface” standard
AXI4	AXI, version 4
BCH	Broadcast Channel
BDF	Boolean Dataflow
CAL	CAL Actor Language
CCF	Component Composition Framework
CCH	Control Channel
CCS	Calculus of Communicating Systems
CD	Channel Decoder
CE	Channel Encoder
CER	Canonical Encoding Rules
CFR	Crest-Factor Reduction
CHREC	(National Science Foundation) Center for High-Performance Reconfigurable Computing
CI	Component Interaction (Ptolemy domain)
CP	(OFDM) Cyclic Prefix
CPU	Central Processing Unit

CQI	Channel Quality Indicator
CSDF	Cyclo-Static Dataflow
CSP	Communicating Sequential Processes
DAG	Directed Acyclic Graph
DDF	Dynamic Dataflow
DDR	Double Data Rate (RAM)
DDS	Direct Digital Synthesis
DFE	Digital Front-End
DFT	Discrete Fourier Transform
DL	Downlink
DMA	Direct Memory Access
DPD	Digital Pre-Distortion
DPN	Dataflow Process Network
DRC	Design Rule Check
DSE	Design Space Exploration
DSL	Domain-Specific Language
DSML	Domain-Specific Modelling Language
DSP	Digital Signal Processing
DUC/DDC	Digital Up-Conversion/Digital Down-Conversion
ECN	ASN.1 Encoding Control Notation
EDA	Electronic Design Automation
EDK	Xilinx Embedded Development Kit
EIDL	Effective Interface Dimension List
eNodeB	Evolved Node B (3GPP LTE equivalent of a GSM base station)
EPIC	Explicitly Parallel Instruction Computing
e-UTRAN	Evolved Universal Terrestrial Radio Access Network (air interface for 3GPP LTE)
FFT	Fast Fourier Transform
FIFO	First-In First-Out, or a buffer that implements first-in first-out behaviour
FIR	Finite Impulse Response (can also refer to the Xilinx FIR Compiler LogiCORE)
FPGA	Field-Programmable Gate Array
GPP	General Purpose Processor
GPU	Graphics Processing Unit

GUI	Graphical User Interface
HARQ	Hybrid Automatic Repeat Request
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High Performance Computing
HPRC	High Performance Reconfigurable Computing
IBSDF	Interface-Based Hierarchical Synchronous Dataflow
ICH	Indicator Channel
IDF	Integer Dataflow
IDFT	Inverse Discrete Fourier Transform
IDL	Interface Dimension List/Interface Description Language
IDS	Interface Dimension Set
IP	Intellectual Property
IPv4	Internet Protocol, version 4
IR	Intermediate Representation
JHDL	Just-another Hardware Description Language
JVM	Java Virtual Machine
KPN	Kahn Process Network
LLR	Log-Likelihood Ratio
LLVM	Low-Level Virtual Machine
LTE	3GPP “Long-Term Evolution” mobile telephony standard
LWIP	Lightweight TCP/IP
MAC	Media Access Control
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MDSDF	Multidimensional Synchronous Dataflow
MII	Minimum Initiation Interval
MIMO	Multiple Input, Multiple Output
MPI	Message-Passing Interface
MU-MIMO	Multi-User MIMO
NL	Network Language
NRE	Non-Recurring Engineering
OFDM	Orthogonal Frequency-Division Multiplexing

OFDMA	Orthogonal Frequency-Division Multiple Access
OMG	Object Management Group
PBCH	Physical Broadcast Channel
PCFICH	Physical Control Format Indication Channel
PDSCH	Physical Downlink Shared Channel
PER	ASN.1 Packed Encoding Rules
PHICH	Physical Hybrid Indication Channel
PIM	Platform-Independent Model
PMCH	Physical Multicast Channel
PRACH	Physical Random Access Channel
PSM	Platform-Specific Model
PUCCH	Physical Uplink Control Channel (can also refer to the Xilinx PUCCH Receiver LogiCORE which implements the processing functions for this channel)
PUSCH	Physical Uplink Shared Channel
QAM	Quadrature Amplitude Modulation
QoR	Quality of Results
QPSK	Quadrature Phase Shift Keying
RAM	Random-Access Memory
RB	Resource Block (in LTE resource grid)
RBG	Resource Block Group (in LTE resource grid)
RDL	Repetition Dimension List
RDS	Repetition Dimension Set
RE	Resource Element (in LTE resource grid)
RGB	Red, Green, Blue
RS	Reference Symbol
RTL	Register Transfer Level
RVC-CAL	Reconfigurable Video Coding variant of CAL
RX	Receive
SCF	System Coordination Framework
SC-FDMA	Single-Carrier Frequency-Division Multiple Access
SCH	Shared Channel
SDF	Synchronous Dataflow
SID	Xilinx Streaming Interleaver/Deinterleaver LogiCORE

SMT	Satisfiability Modulo Theories
STL	C++ Standard Template Library
SU-MIMO	Single-User MIMO
TB	Transport Block
Tcl	Tool Command Language
TDM	Time-Division Multiplexing
TDP	Targeted Design Platform
TPD	Turns Per Day
TT2	(Perl) Template Toolkit 2
TTM	Time To Market
TX	Transmit
UE	User Equipment
UL	Uplink
UML	Unified Modelling Language
VHDL	VHSIC Hardware Description Language
VHLS	Vivado High-Level Synthesis
VIPI	Vivado IP Integrator
VLIW	Very Long Instruction Word
VLNV	Vendor, Library, Name, Version
XFFT	Xilinx FFT LogiCORE
XML	Extensible Markup Language
XPS	Xilinx Platform Studio
XSLT	Extensible Stylesheet Language Transforms
YAML	YAML Ain't Markup Language
ZP	Zero-Padding

Chapter 1

Introduction

For a large part of the history of computing, process scaling according to Moore's Law has led to regular and predictable increases in the performance of single-threaded general-purpose processors (GPPs) through exponential increases in clock rates. However, as a result of problems in areas such as power dissipation and transistor speed, these clock rate increases have reached a practical limit [1]. This is problematic for a wide variety of computing applications that continue to be compute-bound, for example in high performance computing (HPC), and so the continuation of process scaling has been exploited instead to produce parallel multi-core devices in which a standard microprocessor core is replicated a number of times on a semiconductor die.

Aside from achieving raw processor performance, however, there is an increasing need for processors to achieve high levels of performance per watt of power consumption. As HPC systems continue to push the boundaries of the current processor technology, power usage in data centres and supercomputers has continued to grow [2], with the highest-ranked system on the TOP500 list of supercomputers consuming 7.89 megawatts of power [3]. In an attempt to encourage *sustainable supercomputing*, the TOP500 list of supercomputers has recently been supplemented with the Green500 list, ordered by power efficiency [4], and this indicates an awareness within the HPC industry of the large power requirements and resulting running costs of modern architectures and of the desire for power consumption to be reduced.

Various solutions to the problems of performance and power efficiency have been proposed, many of which depart significantly from conventional microprocessor architecture. One approach involves a technique known as *inexact computation*, in which microprocessors are designed such that they allow some degree of imprecision in computed values. This has been demonstrated to provide a 15-fold improvement in area-delay-energy product [5], equivalent to six years of the growth previously expected under Moore’s Law, but provides a new set of challenges in dealing correctly with inexact values.

To address both the raw performance problem and the performance-per-watt problem while retaining exact computed values, other researchers have investigated the use of accelerator devices that are specialized to a particular task. Acceleration using Graphics Processing Units (GPUs) is becoming common in the HPC domain, since they aggregate a large number of processing pipelines in a low-cost device and are becoming increasingly programmable. Another approach, potentially offering even higher levels of performance in many applications, involves devices containing *configurable logic* such as the Field Programmable Gate Array (FPGA) in which the data path may be configured at a fine level of granularity to suit a particular application. It is the production of systems to be programmed to these devices that will be the subject of the investigations in this thesis.

1.1 Configurable logic

In the general semiconductor domain, the fabrication of application-specific integrated circuits (ASICs) is becoming more expensive due to exponentially increasing mask costs [6]. FPGA devices from companies such as Xilinx and Altera provide a means to avoid this expense by providing flexibility (in terms of reprogrammability and time-to-market) while maintaining high performance (in terms of number of operations executed per second), and FPGAs have therefore been adopted as ASIC replacements as part of a trend known as the “programmable revolution” (or, indeed, the “programmable imperative”) [7]. FPGAs are adopted in this role despite the performance cost of programmability

[8] because of the shorter time-to-market and lower non-recurring engineering costs of FPGA-based systems.

Recently, research into the use of customised processing pipelines implemented on configurable logic to accelerate the type of computation performed in standard microprocessor devices has led to the field of *configurable computing* [9]. Since these pipelines are typically implemented on devices that can be configured repeatedly, this is often known by the more common term of *reconfigurable computing*. This has attracted attention in the HPC domain since it has been demonstrated that FPGAs can offer improvements of multiple orders of magnitude over GPPs in terms of computational power per watt in certain applications [10, 11]. Applied in the HPC domain, this technique has become known as *high-performance reconfigurable computing* (HPRC).

Studies to compare the relative performance of GPPs, GPUs and FPGAs tend to report that GPUs perform better than GPPs, and FPGAs perform better than GPUs. For example, one study demonstrated a speedup of 50x on GPUs and 162x to 544x on FPGA, depending on the precision and variability of precision (fixed or floating point) of the calculations performed [12].

The reasons for the performance disparity arise from the different models of computation used by each device. The *von Neumann architecture* of a typical microprocessor uses a model of computation based on the sequential manipulation of state: it repeatedly loads an instruction from memory, loads data from memory, performs some operation such as an addition or multiplication, and writes the result back to memory¹. Modern examples of this architecture are designed to deal with unpredictable control flow, with instruction streams containing frequent jumps, conditional branches and subroutine calls. This limits the length of the processing pipelines and requires significant amounts of logic to be dedicated to tasks such as branch prediction and out-of-order execution. Furthermore, the frequent need to load and store data through a narrow processor-memory interface leads to a phenomenon termed the *von Neumann bottleneck* [13].

GPUs, in contrast, target applications with more predictable control flow, and this allows

¹Modern out-of-order GPPs do not execute code in a strictly sequential manner, but they are bound by the requirement to *appear* as though they do.

control logic to be replaced with long dataflow pipelines that, through the interleaving of processing and register storage, avoid the memory bandwidth problems encountered in von Neumann architectures. Furthermore, these pipelines are replicated in parallel, allowing high performance in applications with high levels of parallelism. While the GPU is still a reasonably general-purpose device with the structure of each pipeline being fixed, the ability to create pipelines in an FPGA that are customised to an application allows smaller area requirements per pipeline, and in turn this allows a larger number of pipelines to be implemented in each device. Thus, even greater levels of peak performance may potentially be achieved in an FPGA.

1.2 Heterogeneous platforms

Wholesale adoption of FPGAs to replace GPPs is not possible, because FPGAs (and GPUs) are only useful for certain types of computation. While software applications that make heavy use of highly parallel or regular signal processing techniques such as Fourier transforms would, in many cases, perform better if this processing were performed by an optimised FPGA core, FPGA-based applications often made effective use of an embedded microprocessor to handle sporadic or irregular data processing patterns. Thus, there is a need for different types of computation to be targeted by different processing styles on a heterogeneous platform [14].

To address the need for tight integration of software and FPGA processing, FPGA vendors provide *soft core* processors such as the MicroBlaze [15] and Nios II [16]. To provide improved software performance, FPGA devices have recently been manufactured which contain hard microprocessor cores. Devices from Xilinx have included the Virtex-5 FXT with an embedded PowerPC core [17], and more recently, the Zynq-7000 series of *extensible processing platforms* with two powerful ARM Cortex-A9 MPCore application processor cores [18]. Altera is following a similar path, with a partnership with Intel leading to the E6x5C series of Atom processors with FPGA fabric and the development of a *SoC FPGA* that incorporates ARM cores [19]. In the evolution of these architectures, a shift may be observed from the use of microprocessors as secondary components in an otherwise

standard FPGA to the addition of FPGA fabric to a standard microprocessor.

1.3 Programming abstractions

A challenge faced by vendors of all programmable devices, whether GPPs, GPUs or FPGAs, is to develop an ecosystem of supporting tools that allow high performance applications to be created with as little effort as possible for the devices they produce.

GPPs were originally programmed using low-level assembly languages which mapped instructions in input code directly to operations performed by the functional units of the device. Gradually, it became possible through advances in compiler technology to widen the *semantic gap* between programming constructs and processor operations, and high-level languages such as Fortran and C became standardised to the point that any performance gains from the use of assembly languages were outweighed by the ease of expression afforded by the new languages. Development of language paradigms has continued, with object orientation allowing greater code re-use, extensibility and modularity, and functional programming allowing the declarative description of algorithms in a mathematical style. These advances allow software programmers to construct working programs with minimal knowledge of the underlying hardware and to delegate performance optimisation to compilers.

However, these abstractions are built upon the assumption of an underlying von Neumann execution model, and though modern *superscalar* processor microarchitectures have become increasingly parallel, much of this parallelism is inexpressible by the sequential instruction sets that are typically implemented on top for reasons of backwards compatibility [20]. Where parallelism may be specified at the instruction level, such as in Very Long Instruction Word (VLIW) and Explicitly Parallel Instruction Computing (EPIC) architectures, the problem of determining valid parallel instruction sequences is shifted from the processor to the compiler, but finding enough parallelism in the sequential input code remains a challenge. Similarly, the move towards multi-core processors has occurred ahead of the necessary advances in languages and tools needed to exploit the parallelism that the new architectures provide [21].

In contrast, the hardware description languages (HDLs) that are used in the process of FPGA system design allow the expression of fine-grained parallelism, but only at a low level of abstraction; an engineer writing register transfer level (RTL) code in a hardware description language (HDL) must remain aware of the physical layout of the target FPGA in order to produce an implementation of sufficient speed. Reasons for this include the relative immaturity of the domain, the requirement for “thinking in parallel” at all stages of the design process, and the need for hardware engineers to extract maximal levels of performance from their designs in order for the FPGA to remain cost-effective as a microprocessor or ASIC alternative. As a result, FPGAs are regarded as being significantly harder to program than GPPs with the current state-of-the-art design tools.

There is not yet a consensus on the best tools and techniques with which to widen the language bottleneck in the software domain, and to bridge the gap between algorithms and RTL code in the hardware domain. On the software side, one of the most prevalent models used to address the rise of multi-core processors is multithreading, but this can introduce race conditions and nondeterminism in subtle ways that make it difficult for programmers to ensure that their programs are bug-free [22]. On the hardware side, *high-level synthesis* (HLS) tools such as Catapult C [23] and Vivado HLS [24] are available to raise the abstraction level of the hardware design process, but their relative success stands in contrast to the failure of a number of other tools in this domain such as AccelDSP [25], the Mitrion Virtual Processor [26] and Handel-C [27] to provide sufficiently high quality of results to benefit from significant market penetration.

The differences between the parallel programming difficulties in software and hardware are summarised in Figure 1.1: in software, the ability to design parallel systems is constrained by sequential mechanisms for design expression (instruction sets and most programming languages), while in hardware, a parallel interface is exposed to the user but there is a lack of mature high-level tools that allow algorithms to be targeted easily to that interface.

One way to deal with the complexities of parallel programming in both the software and hardware domains is for experts to produce libraries of parallel designs to stringent performance specifications that are intended to be instantiated easily and reused by cus-

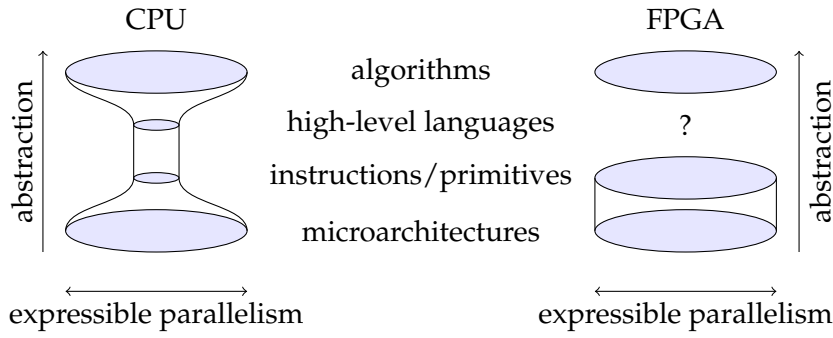


Figure 1.1: Visualisation of the available parallelism at various abstraction layers for two different architectures.

tomers within their system designs. In the FPGA domain, these designs are known as *IP cores*, and they allow high performance FPGA systems to be implemented more easily by hiding the task of specifying behaviour at a low level of abstraction from the end-user, leaving only the task of coordinating the cores.

Xilinx produces a variety of IP cores, which are branded as *LogiCOREs*. Some of these are targeted to specific markets such as wireless communications or video processing, some are used in a broader signal processing domain, including “horizontal” cores such as the FIR filters that are generated by the FIR Compiler LogiCORE [28], and some provide a base platform by fulfilling roles such as Ethernet media access control and DDR memory control [7]. While the task of constructing a system using these cores is ordinarily delegated to customers, recent market dynamics have caused this task additionally to be brought in-house; in Xilinx, this is done through the construction of *Targeted Design Platforms* (TDPs) on heterogeneous processing platforms in order to further stimulate demand in specific markets such as telecommunications and video processing [7].

While IP integration offers significant benefits in FPGA system design, the growth in size of FPGA devices has occurred faster than their tools’ ability to deal with the complexity of this task, leading to a “design productivity gap” [29]. Limited design productivity, defined as the time to complete a new design, the time to do something new with an existing design, and the rate at which a series of designs can be created [30], hampers the speed of development of these systems, and in the context of FPGA systems, this acts as a barrier both to the efficient design of systems for FPGA-based platforms, and to the migration of processor-based systems to hybrid processor-FPGA platforms.

1.4 Goals

The potential benefits of reconfigurable computing in producing high-performance systems are clear, but Xilinx and other companies have a growing requirement for tools that enable fast design and specification of parallel systems on these platforms, particularly those involving the interactions of cores in the FPGA fabric and software components running on a microprocessor. Thus, rather than focusing on performance improvement, the aim of this thesis is to improve design productivity whilst maintaining the performance characteristics that are achievable through status quo design flows.

This problem is tackled within the context of the Xilinx tool suite, and the proposed improvements are demonstrated through the generation of commercially relevant wireless communications systems that are described in the next chapter.

1.5 Contributions

The contributions of this thesis are to describe the implementation of a high-level design methodology for FPGA systems that builds upon industrial practice and which incorporates the following main features:

1. an extensive metadata schema, implemented as extensions to the industry-standard IP-XACT schema [31], which describes the details of the interfaces and operation of Xilinx IP cores;
2. an optimisation stage that may be applied to a high-level description of a multidimensional streaming communications system that performs automatic design space exploration to determine more efficient memory utilisation and latency characteristics; and
3. a demonstration of the automatic generation of software system models comprising models of individual cores from the IP-XACT-based schema developed in point 1, and the generation of extended IP-XACT descriptions from high-level inputs in order to provide an end-to-end toolflow.

Additionally, the contributions of this thesis provide a number of opportunities for further research. For example, the production of high-level descriptions of FPGA systems allows the generation of the software-hardware communications infrastructure required in systems for heterogeneous platforms, and this has been addressed by another research engineer, Stefan Petko.

The solutions proposed in point 2 above have led to the filing and issuing of US patent #8,365,109 [32], which indicates the commercial relevance of the project to Xilinx, and the developments in points 1 and 3 have been published in two papers in international IEEE conference proceedings, indicating that rather than being specific to Xilinx, the problem is of wider academic interest. The first of these papers motivates and describes at a high level the use of IP-XACT as an *intermediate representation* in a code generation flow [33], and the second paper describes in more detail the extensions to IP-XACT that are of most use in enabling the generation of as much low-level software code as possible, and how this downstream generation is achieved [34]. The papers are included in this thesis in Appendix A, and the aim of the remainder of this thesis is to contextualise and explain the published developments in sufficient further detail that the results may be replicated and used as the basis for future work.

1.6 Thesis outline

Chapter 2 describes in greater depth the industrial context of the problem, focusing on the issues encountered within Xilinx in integrating IP cores into large systems. Chapter 3 then examines the techniques proposed in the academic literature to determine whether existing work may be adopted or adapted to solve these problems, and after determining the missing features, Chapter 4 outlines an approach to address these problems that includes metadata, optimisations and automatic code generation. Following this are three chapters that describe in detail the novel features of the approach: Chapter 5 describes the metadata extensions that are used, Chapter 6 describes the optimisations, and Chapter 7 describes the automated code generation flow. Finally, Chapter 8 provides an evaluation, conclusion and discussion of future work.

Chapter 2

Challenges in designing signal processing systems

The recent development of the 3GPP Long Term Evolution (LTE) mobile telephony standard has stimulated demand for compliant component and system implementations, and Xilinx has been working to address this demand with the production of a portfolio of LogiCOREs and system reference designs. This chapter will describe the 3GPP LTE e-UTRAN physical layer (with reference mainly to Rumney et al. [35]), and will then describe the system design challenges that it presents, how these are currently addressed in Xilinx, and some of the limitations of the current approach.

2.1 Xilinx LTE baseband systems

The physical layer of an LTE system encompasses both baseband components, defined by the LTE standard, and digital front-end (DFE) components such as digital up/down-conversion (DUC/DDC), crest factor reduction (CFR) and digital pre-distortion (DPD) which may be implemented for economic reasons: for example, DPD allows less expensive power amplifiers to be used in base stations. There are four distinct LTE baseband systems: downlink transmit, downlink receive, uplink transmit and uplink receive. The downlink receive and uplink transmit systems are found in user equipment (UE) such as

mobile phone handsets, while the downlink transmit and uplink receive systems belong to the base station (eNodeB) design. The need for four separate systems is motivated firstly by the differing data rate requirements of uplink and downlink, and secondly by the differing power requirements of coding and decoding algorithms: more sophisticated decoding techniques are permitted in the eNodeB than in the UE due to the availability of mains electricity rather than battery power. Since the eNodeB handles the communication from a number of UE devices, peak data rates are higher and the use of FPGAs in eNodeB devices becomes economical. Thus, it is the uplink receive and downlink transmit systems that are of interest to Xilinx, and it is these systems that will be examined in this thesis.

The major tasks that are performed in LTE transmit systems are channel coding, modulation, MIMO encoding, resource mapping and channel multiplexing (OFDMA/SC-FDMA), and the receive systems perform essentially the inverse operations. Figure 2.1 shows the structure of the Xilinx baseband LTE systems along with the DFE and MAC (media access control) context in which they are used. LogiCOREs are used where available to fulfil the processing requirements of the systems, and where no LogiCORE is available, for example in the case of the resource mapper and demapper, custom blocks are implemented as required in the system design process.

Data is communicated in separate channels, with users sharing data transmission capacity in the Physical Downlink Shared Channel (PDSCH) in the downlink, and the Physical Uplink Shared Channel (PUSCH) in the uplink. Control information is communicated in the PDCCH and PUCCH channels, and a number of additional channels are defined for purposes including broadcast (PBCH), multicast (PMCH), hybrid indication (PHICH) and control format indication (PCFICH) in the downlink, and random access (PRACH) in the uplink. In the Xilinx downlink transmit system, similarities in the encoding process allow a common processing chain to be used for the PCFICH and PHICH channels (in an “ICH” encoding chain), and also for the PBCH and PDCCH channels (in a “CCH” encoding chain).

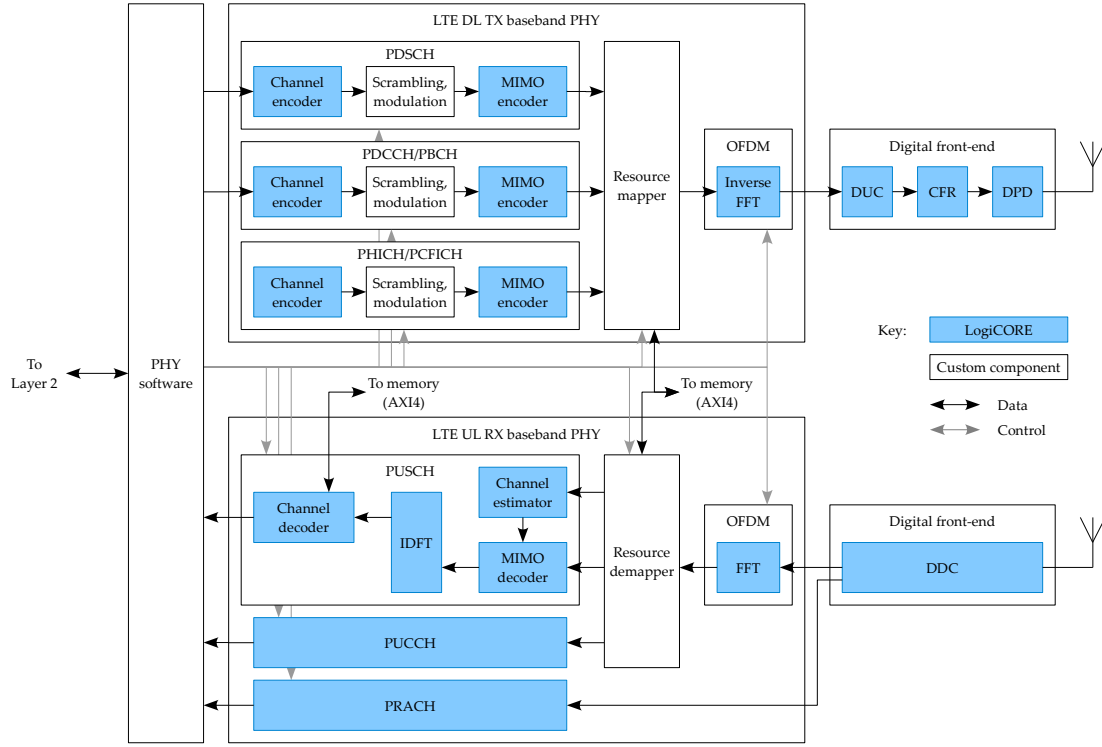


Figure 2.1: The Xilinx LTE downlink transmit and uplink receive systems.

2.1.1 Coding, modulation and MIMO

Each channel undergoes a different coding and modulation process. In the coding stage, data in each channel is segmented into code blocks and a CRC is calculated and appended to each code block with different parameters on each channel. Forward error correction is then applied, consisting of Turbo coding for the shared channels and tail-biting convolutional coding for the main control channels, with other forms of coding applied to the remaining channels. Code blocks are then rate-matched and concatenated. All of these processes are applied by the LTE DL Channel Encoder LogiCORE, which may be configured to perform the required processing for any of the appropriate downlink channels, and the inverse processes in the uplink system are applied by the LTE UL Channel Decoder LogiCORE for the PUSCH and the LTE PUCCH Receiver LogiCORE for the PUCCH. The Channel decoder also implements hybrid automatic repeat request (HARQ): when errors are detected in PUSCH code blocks and cannot be corrected, the receiver requests blocks of *incrementally redundant* data to decode the original signal. While the reuse of previously-transmitted signals in a HARQ process reduces UE power re-

quirements when compared to non-hybrid ARQ (which is used in LTE Layer 2 when HARQ processing fails), the storage of multiple full packets requires large amounts of memory bandwidth in the uplink receive system.

The next stages in the downlink are scrambling and modulation, which are custom blocks. The scrambler flattens the power spectrum of the coded data, while the modulator assigns data symbols to complex-valued points in a constellation diagram that are conveyed on a carrier signal. The modulator applies a different modulation scheme to each channel, and in some cases the scheme is configurable: in the case of the PDSCH, it is determined by a channel quality indicator (CQI) in a process known as Adaptive Modulation and Coding (AMC): for higher-quality channels, it is desirable to transmit a greater number of bits per symbol using a scheme such as 64QAM, while in poorer-quality channels, a 16QAM or QPSK scheme may be used. In the receive systems these processes are reversed by a descrambler and a demodulator, which assigns received data to constellation points by calculating log-likelihood ratios, and these functions are provided in the PUSCH by the LTE UL Channel Decoder LogiCORE.

The purpose of MIMO coding is to exploit *spatial multiplexing* through the use of multiple transmit and receive antennas to improve the capacity and range of radio channels. Each transmit antenna is associated with a *codeword*, and the antennas can be used in either a *single-user MIMO* (SU-MIMO) configuration, in which both codewords are used for a single UE, or in a *multi-user MIMO* (MU-MIMO) configuration, in which a different codeword is used for each UE. Initial releases of the LTE standard focused on SU-MIMO, but in order to meet the demands for greater spectral efficiency in the “LTE Advanced” standard, successive releases have included greater support for MU-MIMO [36].

In the Xilinx LTE systems, MIMO processing is performed by the MIMO Encoder LogiCORE in the downlink transmit system and the MIMO Decoder LogiCORE in the uplink receive system. The MIMO Decoder calculates estimates of the transmitted values for each codeword based on properties of the MIMO channel over which the values were transmitted. The properties of the channel, represented in a *channel matrix* (H) and a noise sample signal (σ), are estimated by a channel estimator component based on known reference signals (RS) that are transmitted regularly, and in the Xilinx LTE systems, the

reference signals are generated by the resource mapper component and the channel estimation is performed by the LTE Channel Estimator LogiCORE. In the resource mapper, synchronisation signals are also generated in order to assist UE devices in synchronizing with an eNodeB. In the next section, the process applied by the resource mapper to map the various channels onto the physical channel resource will be described.

2.1.2 Channel multiplexing and multiple access

LTE makes use of Orthogonal Frequency Division Multiplexing (OFDM), which splits the carrier frequency band into a large number of subcarriers with a low symbol rate, such that the available physical resource is split both into time symbols and into frequency subcarriers. The various downlink and uplink LTE channels must be multiplexed onto these resources, and multiple users must further contend for those resources that are allocated to the shared channel. In the downlink, rather than allocating users to individual subcarriers, they are allocated in both a frequency-division and in a time-division multiplexed manner, so that a user occupies a variety of subcarriers and time symbols at any time. This is known as Orthogonal Frequency Division Multiple Access (OFDMA). In the uplink, a variant known as Single-Carrier Frequency Division Multiple Access (SC-FDMA) is used, which requires an additional inverse DFT to be applied.

The mapping of symbols and subcarriers to LTE channels is determined by a *resource grid*, and there are different grids for the downlink and uplink LTE standards. An outline of the uplink variant is shown in Figure 2.2. This figure shows a single sub-frame of data for a particular uplink configuration, which is divided in the time domain into two slots, each containing seven symbols, of which six are data symbols and one contains a reference signal, and in the frequency domain into a number of OFDM subcarriers, and in the spatial domain into four separate blocks of data received from four different receive antennas. These numbers vary depending on the configuration of the system. Each atomic element within this resource grid is known as a resource element (RE), and REs are grouped into resource blocks (RBs). In a given subframe, data for a particular user may be mapped to a number of resource blocks, and in this thesis these will be referred to as resource block groups (RBGs).

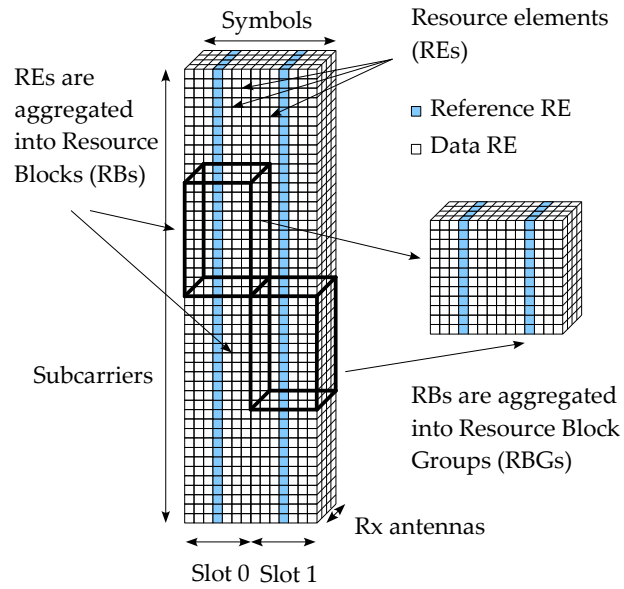


Figure 2.2: The LTE uplink resource grid.

The process of mapping and demapping channel data is performed by a resource mapper in transmit systems and a resource demapper in receive systems. This is done by storing the data for each subframe in DDR memory, accessed through a memory controller, with the resource mapper and demapper generating addresses for each channel corresponding to the position of the data in that channel in the resource grid. In transmit systems, a full subframe of data must be received before OFDM processing may occur, but in receive systems, channels that only occupy earlier symbols in the subframe may be output sooner from the demapper block.

Finally, the OFDM blocks consists mainly of FFT processing, with a forward FFT in the receive system and an inverse FFT in the transmit system. These functions are implemented by the Xilinx XFFT LogiCORE, configured with a separate channel for each antenna in the system configuration. In addition to the FFT, the addition of a guard interval between the OFDM symbols is required to prevent the variance in path lengths in a multi-path channel (caused by reflection from buildings, for example) from causing inter-symbol interference. The guard interval contains a repetition of the symbol data known as a cyclic prefix, and this allows channel estimates to be determined through circular convolution (a simple multiplication in the frequency domain) rather than a more complicated linear

convolution. In the downlink OFDM block, a cyclic prefix is added after the FFT, while in the uplink block it is removed beforehand.

2.2 Challenges in LTE system design

In the process of designing LTE systems, it is desirable for as many of the design processes to be automated as possible. However, some characteristics of the LTE systems and the cores that must be integrated into those systems necessitate a manual approach to system design. These characteristics will be described in the following sections.

2.2.1 Automatic integration of IP cores

The cores that comprise Xilinx LTE systems were designed to provide specific processing functions to top-tier customers, and due to the varying system contexts in which they were to be instantiated, different choices were made in the design of their interfaces. Recently, demand for integrated LTE solutions has increased and it has become desirable for Xilinx to demonstrate the interconnection of the LTE cores in full system designs. Due to the core interface differences, it is inevitable that some additional interface shims must be implemented in to order to connect the cores, and whilst the manual implementation of these shims is feasible, it requires some time and it would be preferable for this to be automated.

In order to automate this task, the interface differences must be understood and classified, and there are a number of aspects to be considered. One of these is the protocol paradigm that is implemented, and within the domain of Xilinx DSP and wireless cores, there are two important paradigms: the cores and custom components in systems tend to have streaming data interfaces¹, while control interfaces are either streaming or memory-mapped.

Associated with each protocol paradigm are a number of defined interface standards. As part of the “Plug & Play IP” initiative, Xilinx cores are standardising on version 4 of the

¹With the exception of the LTE downlink transmit subframe memory controller.

Table 2.1: Interface characterisation of Xilinx DSP cores

Core ^a	Data interface(s)		Control interface(s)	
	Protocol ^b	Standard	Protocol ^b	Standard
DFT	S	Pre-AXI	S	Pre-AXI
DDS	S	AXI4-Stream	S	AXI4-Stream
XFFT	S	AXI4-Stream	S	AXI4-Stream
FIR	S	AXI4-Stream	S	AXI4-Stream
DUC/DDC	S	AXI4-Stream	MM	AXI4
Channel Estimator	S	AXI4-Stream	S	AXI4-Stream
MIMO Decoder	S	Transitional AXI ^c	S	Transitional AXI ^c
MIMO Encoder	S	Pre-AXI	S	Pre-AXI
Channel Decoder	S	AXI4-Stream	Both	AXI4-Stream, AXI4
Channel Encoder	S	Pre-AXI	MM	Pre-AXI
PUCCH	S	AXI4-Stream	S	AXI4-Stream

^a **DFT** : Discrete Fourier Transform v3.1; **DDS** : DDS Compiler v5.0; **XFFT** : Fast Fourier Transform v8.0; **FIR** : FIR Compiler v6.3; **DUC/DDC** : DUC/DDC Compiler v2.0; **Channel Estimator**: 3GPP LTE Channel Estimator v1.1; **MIMO Decoder** : 3GPP LTE MIMO Decoder v2.1; **MIMO Encoder** : 3GPP LTE MIMO Encoder v2.0; **Channel Decoder** : LTE UL Channel Decoder v3.0; **Channel Encoder** : LTE DL Channel Encoder v2.1; **PUCCH** : LTE PUCCH Receiver v1.0.

^b S: streaming; MM: memory-mapped

^c Multiple DATA ports per interface.

Advanced Extensible Interface (AXI) family of interfaces, which provide one streaming protocol (AXI4-Stream) and two memory-mapped interface protocols (AXI4 and AXI4-Lite) which differ in terms of their resource requirements. An overview of the interfaces on a selection of Xilinx cores is shown in Table 2.1.

With the adoption of AXI, cores are becoming compatible at the interface level. However, the presentation of the data on these interfaces is different for each core: in other words, the data types differ between cores. Currently, these data types are described in data sheets in a human-readable format which can be ambiguous, and thus it can be difficult to extract a data arrangement that the core will understand. Furthermore, AXI does not address the tolerance of latency on the interfaces of cores. For example, while the MIMO Decoder will accept control and data transactions on its input ports at any

time, the MIMO Encoder requires these transactions to be simultaneous, while the Channel Estimator cannot receive transactions on its control and data ports simultaneously, and requires them to be sequential. Since these differences are not stored in a computer-readable format, automatic integration flows cannot be implemented.

2.2.2 Design space exploration and optimisation

In the LTE uplink receive system, the data in the resource grid is processed by a sequence of blocks in the system that operate sequentially over different dimensions of this resource grid data, and this prevents them from being connected together directly. One example of this is shown in Figure 2.3: data is output from the MIMO Decoder grouped in a sequence of codewords (cw), and it then undergoes an Inverse Discrete Fourier Transform (IDFT) which is applied across a group of subcarriers (sc) for a single codeword².

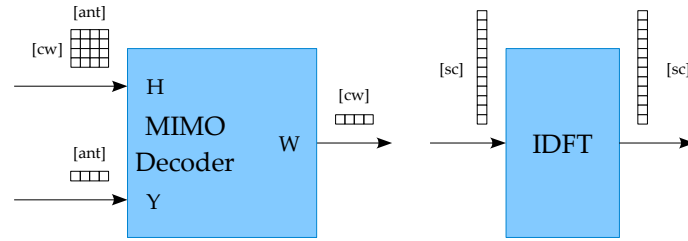


Figure 2.3: A data type interoperability issue in the LTE uplink receive system, caused by different orderings of array dimensions.

These interoperability issues are often solved using data reordering blocks that are implemented manually, such as the one shown in Figure 2.4, but when reorder buffers are required in multiple locations in the data path, it becomes difficult to determine the implications of the placement and implementation of these blocks on the overall memory cost and latency in the system.

Another limitation of the current approach is encountered in the design of the resource demapper, which outputs data from two interfaces: one for reference symbols (REF) and one for data symbols (DATA) as shown in Figure 2.5. The array data types on these inter-

²The MIMO Decoder core may be configured using a `groupsc` control field such that subcarriers are grouped together (i.e. the output data type is `[cw][sc]`), but this discussion assumes that this mode is disabled.

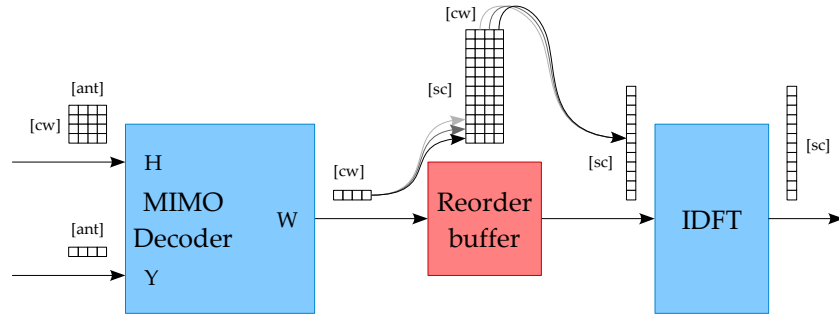


Figure 2.4: Solution to the interoperability issue through the addition of a reorder buffer.

faces are unspecified, but guide the implementation of the resource mapper by determining the order in which it outputs data from the resource grid, and this in turn determines the requirements for reorder buffers downstream. In the absence of extensive metadata, the designer is free to choose from a variety of possibilities, but the validity and efficiency of these possibilities once further development work has occurred cannot be known in advance.

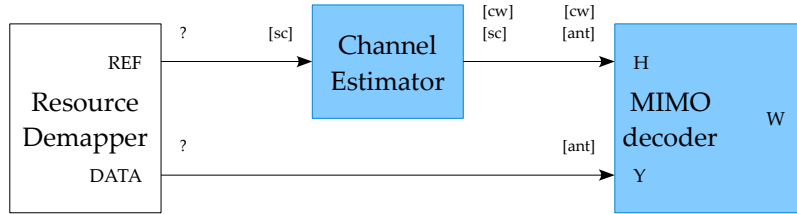


Figure 2.5: Resource demapper output type is dependent on types of neighbours.

Both of these limitations are caused by the need for a bottom-up approach to system design, and improvements could be made if the designer were able to evaluate a number of possible designs in which the configuration, placement or repetition count of components in the processing chain could be altered in order to optimise for desirable criteria such as latency or resource utilisation. This is known as design space exploration (DSE). Performing DSE in the domain of HDL code is laborious due to the low level of abstraction, which causes two main difficulties: firstly, making a high-level change to the system (such as replicating a block and distributing data between the new copies) requires a large number of edits to the code, and secondly, this code requires long compilation times. For systems of non-trivial complexity these issues require the production of a simulation model in order to ensure correctness of the design and to allow high-level modifications

to be assessed rapidly before low-level implementation is carried out.

2.2.3 Software modelling

One of the functions of a software system simulation model is to create test vectors which define the data that should be produced and consumed on the interfaces of each of the subcomponents in the FPGA system, allowing those subcomponents to be implemented in an incremental fashion.

In Xilinx, the modelling and implementation aspects of LTE system development are performed with some degree of independence. The hardware system is implemented by connecting LTE cores and custom components using HDL and system-level design tools, while the software model is implemented in C++ by combining models of those cores and custom software implementations. There are minor structural differences between the models and the implementations, but broadly they correspond to the structure shown in Figure 2.1.

The software models for the Xilinx LTE systems are constructed using a framework called XMODEL, which was designed in Xilinx Scotland and provides an object-oriented class hierarchy for components together with some pre-defined components such as data sources and sinks. Figure 2.6 shows an example component designed according to this framework. A data matrix class is also provided which allows dynamically-sized multi-dimensional data packets to be defined, and an abstract control packet class is provided which must be sub-classed to describe particular control packets. Since the layout of these packets in software memory will not usually be a bit-accurate reflection of the transactions applied on the interfaces of IP cores, particularly in the case of control packets, encoding functions must be defined for each control packet to describe how the representation in software is converted to an encoded test vector.

In order to allow the transfer of data tokens, push and pop functions are associated with components. Communication of a token from one component to another with the same immediate parent is achieved when the parent component calls a pop function on the source component followed by a push function on the destination component. These

XMODEL C++ header (.h) file

```
#ifndef __EXAMPLE__
#define __EXAMPLE__
#endif

#include "in_type.h"
#include "out_type.h"

namespace xmodel
{
    class example : public xbase
    {
    public:
        ...
        void in_push(const in_type&
                     in_token);
        void out_pop(out_type&
                     out_token);
        const out_type& out_peek();
        const bool out_empty();

    private:
        xfifo<in_type>& m_fifo_in;
        xfifo<out_type>& m_fifo_out;
        xtestnode& m_tn_in;
        xtestnode& m_tn_out;
    };
};
```

XMODEL C++ implementation (.cc) file

```
example::example(const
                  xbase_parameters& xparams)
: xbase(xparams)
{
    ...
}

void example::in_push(
    const in_type& in_token)
{
    m_tn_in(in_token);
    m_fifo_in.in_push(in_token);
    v_process();
}

void example::out_pop(
    out_type& out_token)
{
    m_fifo_out.out_pop(out_token);
    m_tn_out(out_token);
}

const out_type& example::out_peek()
{ return m_fifo_out.dout_peek(); }

const bool example::out_empty()
{ return m_fifo_out.dout_empty(); }
```

Figure 2.6: Example XMODEL component definitions.

pops and pushes are scheduled manually by the system designer, and typically (but not exclusively) consist of a sequence of calls of an empty function on an output port, and if the port is not empty, a token is popped from that component and pushed to the destination.

Some components require data from a number of input ports, and in this case, the push function writes data to a FIFO queue associated with the respective port, and processing is then initiated when data is requested by the pop function. Other components produce data on a number of output ports, and in this case, the processing is initiated instead by the push function and the results are written to output FIFOs. When multiple inputs and outputs are required, FIFOs are placed on all interfaces and a function is implemented which checks whether a sufficient number of tokens has been received on all input ports, and if so, performs processing and writes output tokens to output FIFOs.

To allow test vectors to be produced on each input and output port of every component, a “test node” is also associated with each port, and both the push and pop functions write to this test node.

There are some limitations in the current system modelling approach. One limitation is that adding a new component requires the same information (such as the component name) to be entered in a number of locations. This means that necessary modifications can be forgotten, such as modification of the component name in C++ include guards. Another limitation is that implementation decisions involving the instantiation of FIFOs, activities performed by the push and pop functions, and component scheduling, must be made on a component-by-component basis due to lack of a formal model of component interaction that underlies all of the components. The number of modifications to be made for each high-level change limits the amount of design-space exploration that may be carried out. The scope for *automatic* design-space exploration and optimisation at compile-time is also limited, since optimisations must be applied either by the C++ compiler, or at run-time, at a low level of abstraction. Further limitations are that the test vector encoding and decoding functions are laborious to write, and in the down-link transmit model, some type classes also have data processing methods which are not present in the hardware realisation of the system.

It is desirable for system design, modelling and implementation to be as rapid as possible, and the production of a system simulation model requires additional implementation time. In an iterative development process, the model will also have to be changed after low-level implementation has started. This requires differences between the model and the implementation to be reconciled, which takes time. Ideally, a working hardware system could be created from the same description as the simulation model, with an abstract view permitting the incremental mapping of components to FPGA fabric as required. Once a software model has been constructed, the ability to create a hardware system without full reimplementation is highly desirable.

2.2.4 Heterogeneous processing

So far, only the hardware portion of the LTE system implementation has been discussed; as shown in Figure 2.1, there is also a software driver layer. Two issues are apparent in this scenario: firstly, as with the case of test vector production, data type encoding and decoding functions are required to reformat data for software-to-hardware and hardware-to-software communications. Secondly, efficient DMA communications infrastructure must be integrated to allow the software and hardware parts of the system to communicate. Various types of DMA block may be used, and the data rate characteristics of the system must be used to determine, for each hardware-software interface, the block that meets the data requirements with the lowest resource cost. Other challenges include the identification of channels with similar data rates that may be aggregated such that they can share communications blocks, and the inference of these blocks rather than manual instantiation. These issues are addressed by Stefan Petko.

2.3 Summary

The aim of this chapter was to identify a number of limitations of the current system design methodology used in Xilinx, some of which will be addressed in the remainder of this thesis, and some of which are not directly addressed but must be taken into account in the design of a solution. Those which are addressed are as follows:

IP integration Bottom-up design means that IP cores are designed with slight differences in their interfaces. If the differences were represented in a computer-readable format, cores could be integrated automatically.

System-level DSE Beyond simply producing a system that is functionally correct, it is desirable for it to perform well. High performance systems are produced through a process of design-space exploration (DSE), which must currently be performed manually. If the properties of the components in the system are specified explicitly, automated system-level optimisation processes become possible.

Multiple platforms The same system must be implemented in a variety of different contexts, including a bit-accurate software model, cycle-accurate HDL simulation and HDL implementation; each target platform requires a re-implementation of the system which involves replication of boilerplate code. If it were possible to automate the generation of this code, manual implementation time could be saved.

Additionally, the following limitation is addressed by Stefan Petko and provision is made for this in the remainder of the thesis:

Heterogeneous communications The process of designing systems for heterogeneous platforms is complicated by the need to design appropriate processor-FPGA communications infrastructure for each system.

In Chapter 3, a review of existing tools will be presented with the aim of determining whether these problems are adequately addressed in existing work. Having determined the capabilities of these tools, a tool flow architecture which builds upon these capabilities to address the problems in this chapter will be presented in Chapter 4.

Chapter 3

Background

The problems described in Chapter 2 are instances of broader issues that have been examined to some extent in existing academic literature and industrial practice. This chapter provides a review of the existing techniques and tools with the aim of determining, firstly, the features and approaches that have been demonstrated to be useful in existing design frameworks, and secondly, whether any existing framework fully addresses the problems stated in Chapter 2 such that it may be adopted to solve those problems.

3.1 Platform-based design and IP reuse

The principles of platform-based design are to create modular components with common interfaces, so that new components are compatible with existing ones [37, 38]. In one interpretation, a platform may be viewed as a set of designs that are determined by a set of platform constraints – for example, the set of all C programs determined by the syntax of the C programming language, or the set of all x86 executables determined by the x86 instruction set [39]. The low-level implementation of components such as x86 microprocessors is thus targeted “upwards” in abstraction towards a more abstract platform (the x86 instruction set), and a system of software code is created by refining an abstract design “downwards” so that it consists of components in that platform that are described in x86 machine code. In this way, the platform acts as the intersection point of the top-down

and bottom-up processes of system design.

Viewing platforms as sets of designs, it follows that the purpose of individual design tools and compilers is to implement binary relations representing realisable transformations between these sets. A number of different platforms may be involved in the system design process, with each platform representing a different layer of abstraction, and thus a *design framework* may be viewed as a collection of tools and compilers which implement the transitive closure of these relations, providing a chain of realisable transformations from high-level specification to low-level implementation [39].

A good platform has the same characteristics as a good industry standard: it must both be sufficiently descriptive to describe the existing examples of a desirable member of a platform, and sufficiently prescriptive to exclude undesirable examples. These characteristics correspond, respectively, to the concepts of completeness and soundness in mathematical logic, and tools typically exploit the benefits of platform-based design by focusing on one or the other of these. For example, in the context of IP integration and reuse, attempts have been made to improve the usability of IP cores by prescribing more closely the interfaces on cores [40] and the metrics with which to judge the quality of IP designs, through the ‘Quality IP’ metric of the Virtual Socket Interface Alliance [41]. On the other hand, the creation of a language for describing the differences between IP interfaces allows automatic verification of their compatibility or synthesis of bridges to resolve incompatibilities: the Coral tool verifies connections using binary decision diagrams and synthesises glue logic to link components together [42], and in Xilinx, Paul McKechnie recently designed a type system which can be used to describe the interfaces on IP cores and implemented a type checker which verifies the correctness of the connections between those cores [43]. Synthesis of bridges between incompatible interfaces has also been tackled [44, 45].

In practice, a combination of prescription and description is typically used to derive platforms, with simultaneous evolution of the platforms to support the cores and the cores to target the platforms. Within Xilinx, the need to simplify the process for connecting cores at the interface level has been addressed by prescribing a single bus interface standard across all IP cores, namely AXI. Due to the guaranteed interface-level compatibility of

cores with AXI interfaces, interfaces conforming to the same AXI bus type may be connected without error. As the number of different interface standards used by IP providers is reduced, the number of required interface bridge combinations drops exponentially. At the same time, descriptions of LogiCORE IP and system designs are being created and integrated into Xilinx tools, using a standard called IP-XACT [31].

3.1.1 Interface specification in IP-XACT

By associating metadata with components and systems at the time that they are designed, the time-consuming and error-prone task of determining a component's characteristics from a textual or pictorial description in their datasheets can be avoided. IP-XACT is an industry-standard XML schema that is being adopted by many organisations in the Electronic Design Automation (EDA) industry to represent this metadata [46], and it defines a number of top-level object descriptions, such as component descriptions and design descriptions. Component descriptions can be used to store information pertaining to individual components and design descriptions can be used to represent hierarchical designs consisting of those components. Important features of a component description are shown in Listing 3.1.

Listing 3.1: IP-XACT component description example.

```
<spirit:component>
  <spirit:vendor>xilinx.com</spirit:vendor>
  <spirit:library>ip</spirit:library>
  <spirit:name>xfft</spirit:name>
  <spirit:version>8.0</spirit:version>
  <spirit:busInterfaces>
    ...
    <spirit:busInterface>
      <spirit:name>M_AXIS_STATUS</spirit:name>
      <spirit:busType spirit:vendor="xilinx.com" spirit:library="axi4"
                    spirit:name="AXI4Stream" spirit:version="1.0"/>
      <spirit:slave/>
      ...
    </spirit:busInterface>
    ...
  </spirit:busInterfaces>
  ...
  <spirit:parameters>
    <spirit:parameter>
      <spirit:name>ovflo</spirit:name>
      <spirit:value spirit:resolve="user"
                    spirit:id=PARAM_VALUE.OVFLO"
                    spirit:format="bool">false</spirit:value>
    </spirit:parameter>
    ...
  </spirit:parameters>
  ...
</spirit:component>
```

The description begins with a “VLNV” describing the vendor, library, name and version of the component. A list of busInterface elements then detail the available bus interfaces together with their interface type and directionality, and a list of parameter elements detail the parameters that may be set together with information on whether they are resolved automatically or manually, and the default value: in the ovflo example above, the default value is false.

In some cases, interfaces may be enabled or disabled based on core parameters, and this cannot be described in the base IP-XACT standard. At various points in the IP-XACT schema, vendor-specific extensions may be included to extend the description and so this feature been added within Xilinx in the form of a xilinx:enablement vendor extension element. The M_AXIS_STATUS interface is an example of such an interface, and its presence

condition is recorded as shown in Listing 3.2.

Listing 3.2: Representation of an optional bus interface in IP-XACT using vendor extensions.

```
<spirit:busInterface>
  <spirit:name>M_AXIS_STATUS</spirit:name>
  ...
  <spirit:vendorExtensions>
    <xilinx:busInterfaceInfo>
      <xilinx:enablement>
        <xilinx:presence>optional</xilinx:presence>
        <xilinx:isEnabled
          xilinx:resolve="dependent"
          xilinx:dependency="spirit:decode(id(...))"
        >true</xilinx:isEnabled>
      </xilinx:enablement>
    </xilinx:busInterfaceInfo>
  </spirit:vendorExtensions>
</spirit:busInterface>
```

where the dependency is specified using a function of a number of parameter values specified in the XPath language, using the `id` function to reference parameter value elements and the `spirit:decode` function to convert these into integer values. For example, the `M_AXIS_STATUS` interface will be present when overflow is enabled, and the dependency is stated as follows¹:

```
spirit:decode(id('PARAM_VALUE.OVFLO')) = 1
```

IP-XACT also provides design descriptions. These are used to describe the list of component instances in a system, their configuration, and their interconnections, which may be between ports or between aggregated bus interfaces that consist of a number of ports. An example of a design description is shown in Listing 3.3.

¹In reality, the presence of this interface additionally depends on other user-specified parameters, through a chain of dependent parameters.

Listing 3.3: IP-XACT design description example.

```
<spirit:design>
  <spirit:vendor>xilinx.com</spirit:vendor>
  <spirit:library>ip</spirit:library>
  <spirit:name>A</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:componentInstances>
    <spirit:componentInstance>
      <spirit:instanceName>B-inst</spirit:instanceName>
      <spirit:componentRef spirit:vendor="xilinx.com"
        spirit:library="ip"
        spirit:name="B"
        spirit:version="1.0"/>
    </spirit:componentInstance>
    <spirit:componentInstance>
      <spirit:instanceName>C-inst</spirit:instanceName>
      <spirit:componentRef spirit:vendor="xilinx.com"
        spirit:library="ip"
        spirit:name="C"
        spirit:version="1.0"/>
    </spirit:componentInstance>
  </spirit:componentInstances>
</spirit:design>
```

IP-XACT component descriptions are able to describe the mechanisms used by a component for low-level communication of data streams across interfaces, but the discussion in Chapter 2 indicates that there are advantages in considering the interfaces of AXI-compatible cores at a higher level of abstraction. These benefits may be realised by augmenting the metadata to describe how these data streams are encoded with a particular data type, and to describe the interaction of interfaces and their timing constraints in order to consume and produce data in the correct sequence, but there is no industry-standard format for these metadata extensions. Previous work on a schema known as CHREC XML has proposed that these aspects of interface compatibility may be considered in a layered structure, with the following layers [47, 48]:

- an *RTL layer* describing the component's ports, their direction and width, and their grouping into interfaces²,
- a *data type layer* describing the data types communicated over that interface, and

²Information on low-level interface protocols such as the valid/ready handshake in AXI4 might also be considered as part of this layer, but this information is rarely encoded in metadata.

- a *behavioral layer* describing the component’s latency and necessary delay between the introduction of data tokens (initiation interval).

IP-XACT describes components at the RTL layer, and further extensions would be of benefit in expressing compatibility at the data type layer and interface operation layer. This problem has been partially addressed in CHREC XML, which introduces basic descriptions of the data types communicated on component interfaces [49], together with temporal information such as latency specifications and a design environment called Ogre that uses the CHREC XML extensions to generate wrappers for IP cores [50]. However, the metadata in these proposals is not sufficiently extensive to describe Xilinx IP: for example, data types are described simply in terms of their total and fractional bit widths, and there is no provision for the description of the hierarchical packets that IP cores receive on their control interfaces.

Aside from the matter of describing existing IP cores, there is still the issue of design productivity in the initial creation of new cores, and most existing design flows still involve a large amount of manual effort in this process. Additionally, since systems often need components that are not found in an IP library, it must also be possible to design those components efficiently.

3.2 Designing new components

Typically, FPGA cores and systems are designed on paper and implemented at a low level of abstraction in hardware description languages. To allow more substantial design space exploration, it is often prudent to produce a more abstract model first, explore various designs, and produce a low-level implementation only once an optimised design has been found. The model can be described using existing software languages, or with domain-specific modelling languages (DSML) and metamodeling.

The modelling of electronic systems in software languages is often simplified through the use of object-oriented frameworks such as XMODEL (which is used in Xilinx) and SystemC (which is used more widely in the industry) [51]. One problem with approaches

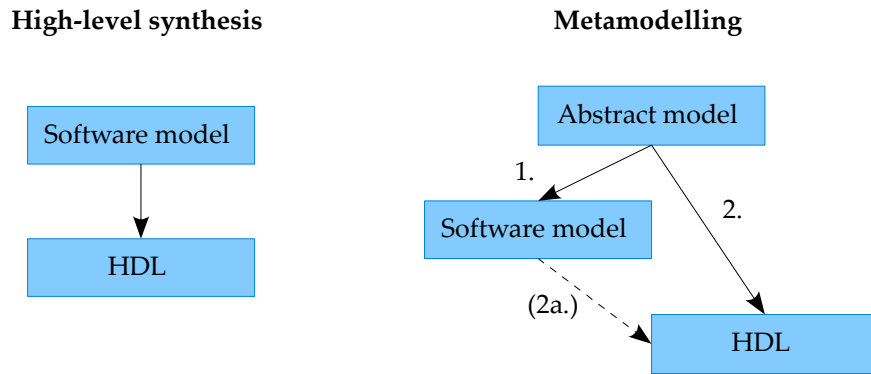


Figure 3.1: High-level synthesis and metamodelling.

based on software languages is the overhead of software engineering, encountered when dealing with error messages caused by misapplication of C++ template syntax, for example. While many C++ experts would find this acceptable, others would argue that detailed knowledge of a software language should not be necessary in the design of hardware systems.

Modelling systems in software also requires the production of a full additional description of the system in a software language rather than a hardware language, and this often entails a significant amount of work that can be avoided if the same system description is used for both modelling and implementation. Two alternatives have been proposed, which are shown in Figure 3.1. The first approach is often called *high-level synthesis* (HLS), and involves the generation of a low-level implementation from a higher-level software representation that may also be used as a simulation model. The second approach, known as *metamodelling*, requires system characteristics to be captured in a single representation that is independent of the properties of the simulation or implementation context, and that representation is used to generate a simulation model or an implementation (or a large part thereof) as required. Optionally, HLS can be used to generate hardware from the software output of a metamodelling flow (arrow 2a in Figure 3.1).

Since the input to an HLS flow is a software representation of a system, it may be readily compiled and executed using existing software compilers and other infrastructure, whereas a metamodelling flow starts with a more abstract representation and requires additional computational specifications to allow execution on target platforms.

3.2.1 High-level synthesis

There are two common approaches to the synthesis of software code into a hardware description. The first involves the modification of a software compiler such that instead of outputting microprocessor machine code, it outputs HDL. This approach is used in tools such as AutoESL, which was recently acquired by Xilinx and is now known as “Vivado HLS” [24] or “VHLS”. The second involves the creation of an object-oriented software framework, effectively creating an *embedded domain-specific language* for the description of hardware systems, and this approach is used in JHDL [52], the Microsoft Accelerator tool [53] and Maxeler MaxCompiler [54]. These system descriptions may then be compiled with a standard software compiler, and the HDL is generated through the execution of the compiled code. Compiler modifications may still be necessary to allow greater syntactic convenience in the host language (as has been done in MaxCompiler through the modification of a Java compiler), but these modifications are less extensive than in the previous case.

Of these approaches, the first presents a shallower learning curve to users with prior knowledge of the targeted software language, since workable (if inefficient) systems can often be created with minimal deviation from standard software coding style; improvements to those systems may then be realised incrementally through the addition of compiler directives or through modification of coding style. However, the semantic gap between software language and hardware implementation means that behaviour must always be inferred in this approach, and such inference does not always correspond with the designer’s intent. The second approach, in contrast, has both the advantages and disadvantages of other approaches based around domain-specific languages, which provide the potential for high quality of results at the expense of a steeper learning curve and (as the name suggests) a more limited domain of applicability.

A typical high-level language trades the expressiveness of a low-level language in describing a wide variety of designs for greater succinctness in describing the most common or useful designs. Thus, it is inevitable that with the use of a high-level input representation, certain design features cannot be implemented as efficiently as in the original

low-level representation. For example, it may not be possible to customise the logic required to perform an addition to a sufficient degree. This has not been a widespread barrier to the adoption of high-level software languages in place of assembly languages, but poorer quality of results (QoR) relative to the status quo remains a common criticism of HLS tools, perhaps unfairly.

One possible explanation for this is that there is a closer correlation between code efficiency and total system cost in hardware designs: in interactive software, inefficiency will often result only in delays that are imperceptible to a human operator, whereas hardware systems often have tight latency constraints and inefficiency is paid for in additional resource requirements which necessitate the use of more costly FPGA devices. However, in many applications, excessive development time and non-recurring engineering (NRE) costs are greater risks to the success of a project than the use of expensive FPGA devices, and in these applications the use of HLS tools is desirable [55].

For other applications, there remain a number of outstanding practical problems with HLS. Firstly, while good QoR has been demonstrated in some scenarios such as sphere decoding [56], IP designers require a substantial body of evidence before they can adopt HLS wholesale. Secondly, many cores and systems are constructed from sub-cores, and in the case of Vivado HLS at least, the lack of data type and interface timing metadata for existing IP cores prevents their integration in HLS designs.

3.2.2 Metamodelling

Rather than using existing software languages, an alternative system design approach uses the characteristics of a problem domain to derive simplified representations of those systems called *models*. Within Xilinx, the term “model” typically refers to a software simulation model, but in the language of metamodelling, the term is used more broadly: indeed, the principle that “everything is a model” has been proposed in the context of metamodelling in the same way that “everything is an object” applies in an object-oriented view of a system [57]. Thus, for a system implemented on an FPGA, the HDL description of the system is also a model of the system since it describes the operation of

the system at a more abstract level than the physical movement of electrons.

Compared to object orientation, metamodeling has a similar notion of a hierarchy of entities in a directed acyclic graph (DAG) structure, but instead of objects being instances of classes, and classes inheriting from superclasses, systems are *represented by* models, and models *conform to* metamodels. Viewed in one light, a metamodel is essentially a description of a platform, and it is the formalisation of these metamodels that characterises the metamodeling approach.

A particularly useful model is one that represents independently and orthogonally the principal concerns of a domain: in other words, the characteristics of the domain that are independent of specific software or hardware implementation technologies [37, 58]. Within the domain of IP cores, IP-XACT provides an appropriate language in which to represent some of these concerns, such as component interfaces and hierarchy, and vendor extensions may be used to represent additional concerns such as the RTL layer, data type layer and behaviour layer information described by CHREC XML. Approaches based on XML can also be used to represent metadata in a reasonably orthogonal manner: for example, the interfaces on components may be modified orthogonally to the topology of their interconnection in an IP-XACT design description, which does not specify the interface types of its component instances. In contrast, these concerns are not orthogonalised in a VHDL description.

There are a number of useful techniques which arise from this view. Model-driven engineering (MDE) is the process of producing a model conforming to a target metamodel from a model conforming to a source metamodel, and is often used in the context of incremental refinement of a model into simulation or implementation code: in other words, the refinement of a platform-independent model (PIM) to a variety of platform-specific models (PSMs). This is achieved through a sequence of model transformations, of which there are two dimensions: endogenous vs. exogenous, and horizontal vs. vertical [59]. The first concerns the language (though within a language such as XML, it might concern the *schema* as a sub-language), and the second concerns the abstraction level. Examples of each transformations in each of the four possible categories are shown in Table 3.1.

Table 3.1: Examples of model transformations.

	Horizontal	Vertical
Endogenous	Refactoring	Formal refinement
Exogenous	Language migration	Code generation

There are two significant advantages of model-driven engineering approaches: firstly, that the total amount of code required to be written is reduced, since the minimal information required to capture the principal concerns can be used to generate the code required in different implementation scenarios, and secondly, that it becomes possible for high-level optimisations to be applied that are difficult to extract from representations of a system that are targeted to a particular execution platform.

One example of an MDE approach is found in the OMG Model-Driven Architecture (MDA) [57, 60], which merits some discussion as one of the most mature examples of MDE. MDA is based around the Unified Modeling Language (UML) [61, 62], and due to the broad scope and terminological complexities of this language, the appeal of MDA within an organisation is likely to depend on the degree to which its use is already ingrained. Since I did not encounter widespread use of UML in Xilinx, I did not consider it in depth in the context of Xilinx methodologies and tool flows. Instead of using UML, some MDE flows have been based on IP-XACT [63, 64], but not in a manner that addresses the problems in Chapter 2, and thus an alternative approach is required.

3.3 Composition of components

High-level synthesis and model-driven engineering deal effectively with the challenge of generating an implementation instance from a high-level description through “vertical” refinement of models through one or more levels of abstraction. However, there is also the challenge of “horizontal” enlargement of systems through the interconnection of a number of component instances, and these systems are often hierarchical and may target a heterogeneous platform. Depending on the information associated with these components, verification of the correctness of the interconnections may also be beneficial.

One option is to describe and coordinate the components' connections using implementation languages such as C and VHDL, both of which permit hierarchical composition: for example, a heterogeneous mix of C functions and HDL cores might be coordinated using a C system description that instantiates the HDL cores using function calls. In this approach, the required data flow in the system is determined implicitly from the language. An alternative is to use an explicit coordination language with minimal semantics [65], which enforces a separation of concerns between computation and coordination. Such a language, which may be either textual or visual, may then be combined with procedures for the analysis and optimisation of the system in a component composition framework (CCF) [66], also known as a system stitching tool.

Various CCFs have been produced, both in academic and industry, and these tools often focus on different aspects of the component composition problem. On the academic side, Balboa is a CCF which provides a C++ framework for constructing components together with two languages to allow their integration at a high level of abstraction [67, 68]. These additional languages fulfil a similar role to, but predate, IP-XACT component and design descriptions. The MCF tool builds on Balboa with metamodelling techniques and allows component metadata to be extracted from SystemC blocks and stored in an XML format [69]. An IP selector is provided which is able to instantiate IP cores based on a number of schemes: for example, based on name and version, or interface type, or whether it is a black box or hierarchical. The GASPARD framework (Graphical Array Specification for Parallel and Distributed Computing) [70] is a UML-based MDE framework in which components are described at a high level with a repetitive, tile-based model of computation in the ArrayOL language [71], and refined to SystemC, OpenMP or VHDL implementation code as required.

Other CCFs are targeted towards the creation of systems on heterogeneous platforms. For example, the System Coordination Framework (SCF) [72] is a CCF for heterogeneous systems that allows rapid exploration of different component mapping possibilities: components are designed in software or HDL to a generic communication interface, then connected using a flexible task graph language with control structures that are particularly appropriate for HPC applications with a regular arrangement of compute nodes. Using

the heterogeneous system topology information, communications infrastructure is then inserted automatically between components on different architectures. However, SCF is not a metamodeling tool, and message-passing primitives such as `SCF_init` and `SCF_send` must be added directly to low-level source code, limiting possibilities for design space exploration.

One of the most advanced industrial CCFs is The MathWorks' Simulink, which is a model-based design environment that allows refinement of models through to simulation or implementation on FPGA. LabVIEW is an example of a similar tool produced by National Instruments. Within Xilinx, the System Generator tool has been built on top of Simulink to target the creation of systems on Xilinx platforms [73], but its restrictive model of computation limits its applicability in the context of AXI systems: IP cores with buffered AXI4-Stream interfaces must be coordinated by a tool that assumes predictable latency behaviour. Other Xilinx tools include the Xilinx Platform Studio (XPS), which is part of the Embedded Development Kit (EDK) [74], and a number of unreleased research tools such as Grouse, Brace and System Stitcher [43].

One problem with many of these tools is that they are not based on industry standard metamodels, which prevents the separation of concerns between computation and coordination. Recently, Xilinx has released the Vivado suite of tools, which make extensive use of the IP-XACT standard: cores are stored in an IP catalog with IP-XACT component descriptions, and the new Vivado IP Integrator tool (VIPI) allows systems to be constructed from these cores and connected either at the level of AXI interfaces, or at the level of individual ports, allowing backwards compatibility with pre-AXI cores [75]. The tool uses the component descriptions to determine the available interfaces and parameters on cores, and saves the system topology in the form of an IP-XACT design description.

3.4 Models of computation

Aside from the matter of connecting blocks together there is the issue of what model of computation they use, which defines the patterns of component interaction in the system. Various models have been proposed, including *process algebras* such as the Calculus

of Communicating Systems (CCS) and Communicating Sequential Processes (CSP), but in the context of signal processing systems, the *dataflow* paradigm has been of particular interest. This term has been applied to computer architectures such as the Manchester dataflow machine and to languages used to program these architectures such as LUCID and SISAL, but due to problems in achieving high levels of performance from these architectures, they have been regarded as “mostly... a failure” [76] and have not gained widespread acceptance. The recent interest in FPGAs as a computing architecture has reignited interest in this area, but a number of refinements of the programming model have been proposed.

3.4.1 Process networks and actor-oriented design

Rather than specifying applications using fine-grained dataflow languages at the level of arithmetic and logical operations, they may be considered as a network of relatively coarse-grained processes. In a Kahn Process Network (KPN), processes are connected through their ports by point-to-point unidirectional FIFO channels with unbounded capacity, and each process consists of an imperative sequence of computations and reads and writes of data tokens to and from ports [77]. An advantage of Kahn Process Networks over multi-threading is that their execution is deterministic: since the processes are *monotonic*, which is to say that they produce output as soon as the necessary inputs are available, the sequence of outputs from each process is dependent only on the sequence of inputs, and thus the order of the data communicated on each channel does not depend on the order of execution of the processes [78].

Another line of research tackles the problem of extracting maximal performance in the execution of a dataflow graph, through the determination of static schedules which remove the need for extensive buffering and frequent process suspension and resumption. In the Synchronous Dataflow (SDF) theory, static token rates are associated with each node in the dataflow graph, and these are used to determine the relative rate at which each node should be executed, which may in turn be used to determine a static schedule [79]. Synchronous Dataflow requires strict conditions to hold on the actors in the system, one of which is that every actor may have only a single set of token rates, and successive

developments of the theory have introduced additional models of computation which offer relaxations of these conditions in exchange for greater difficulty in determining a schedule. These developments include Cyclo-Static Dataflow (CSDF), which allows nodes to cycle between different sets of token rates [80] and which is reducible to SDF [81], Boolean Dataflow (BDF), which allows token rates to be determined by boolean values received on data channels [82] but at the expense of the ability to determine memory allocation statically on those channels [78], and Integer Dataflow (IDF), which is a generalisation of BDF to integer selector tokens [83]. Further developments include Interface-based Hierarchical SDF (IBSDF), which allows the hierarchical composition of actors [84] and Multidimensional SDF (MDSDF) [85] which generalises SDF to target applications with multidimensional data streams. The most widely applicable dataflow domain is Dynamic Dataflow (DDF), in which all scheduling is performed at run-time [78].

The main difference between Kahn Process Networks and Dynamic Dataflow is the need (in general) for a multi-process implementation context in the former, involving either concurrent processing or a pre-emptive task-scheduling operating system, and this is avoided in the latter through the quantisation of processing into a scheduled sequence of *actor firings* [78]. This actor-based view has been used to unify the various dataflow domains under the theory of Dataflow Process Networks (DPN). A DPN is a set of *actors*, each of which is associated with a set of actions that *fire* when their *firing rules* are satisfied, thereby mapping input tokens to output tokens. A sequence of action firings forms a *dataflow process* [78].

In a dataflow process network, the specifications of the communications performed by the actors are separated from those of the computations that are performed, and in so doing, the model of computation becomes a property of individual actors rather than of the system as a whole. While the use of the term ‘dataflow’ is reminiscent of dataflow languages such as LUCID and SISAL, the actors in a dataflow process network use dataflow concepts at a coarser level of granularity, and the layering of these concepts on top of platform-specific, performance-oriented languages like C and HDL allows high-level analysis and optimisation without sacrificing performance at finer levels of granularity. Thus, systems may be created with heterogeneous low-level models of computa-

tion, which in turn orthogonalises the notions of component definition and component composition [39]. The high-level dataflow domains used in a system may also be heterogeneous, with static scheduling being applied to sub-networks in the system where the components' firing rules indicate that this is possible. This has been demonstrated in practice in the Ptolemy tool [86], which also allows the construction of *domain-polymorphic* actors that can operate according to a variety of models of computation depending on the domain in which they are instantiated.

A further advantage of explicit actor properties is that optimisations may be applied with greater ease to the dataflow graph to reduce its memory, communication and throughput requirements. For example, blocks that are known to be stateless may be *folded* and *unfolded* with the use of split and join blocks, and streaming analogues of software optimisations like dead code elimination and constant propagation may also be applied [87].

A variety of frameworks for the construction of process networks have been created, of which one of the most widely-used commercial implementations is SPW [88]. The OpenDF project [89] has been built around the CAL actor language, and provides tools for the simulation of systems described in that language and the generation of hardware implementations [90] via an intermediate representation called XDF. An example of a CAL actor description is as follows:

```
actor add()
  a, b ==> out:

  action [a], [b] ==> [a + b] end
end
```

This defines an actor with two inputs and one output, with an action that consumes a token from each input and produces a new token on the output. The value of the output token is equivalent to the sum of the values in the input tokens. CAL also supports the notions of token *type* (although this is optional, and CAL does not provide a means to define types), actor state and initialisation actions and action guards and priorities, amongst other features [91]. These features will be discussed further in Chapter 7.

CAL is currently being extended through the ACTORS project, and it has also been embraced and extended into the RVC-CAL language in the domain of reconfigurable video

coding [92]. The Open RVC-CAL Compiler (Orcc) has been implemented to generate executable software code from RVC-CAL descriptions.

In some other frameworks, rather than specifying the process network characteristics explicitly, they are extracted from suitably-constrained sequential code. The Compaan tool infers process networks from MATLAB code [93] and uses these to generate networks of hardware cores, and the KPNgen tool in the Daedalus framework can extract this information from a static affine nested loop program (SANLP) described in the C language [94].

3.5 Platform mapping

Once a design has been proposed as a collection of abstract blocks, those blocks must be mapped to an implementation platform. Typically this is carried out implicitly in the production of software or HDL implementation code, but in the case of heterogeneous platforms, there are a number of possible mappings of the components in the application to the different components in the platform, each with different performance and cost characteristics. On heterogeneous platforms, this mapping process is typically done manually, and the Y-chart approach models how this might be done in practice [95].

The ESPAM tool, also part of the Daedalus framework, addresses this issue with three specifications: a *platform specification* describing the topology of a processing platform, an *application specification* describing the application to be executed, and a *mapping specification* describing how components in the application are mapped to processors [96]. It has also been demonstrated how this mapping process could be done automatically [97].

As a framework for the KPNgen, ESPAM and Sesame tools (the latter of which allows high-level design space exploration), it might be assumed that the Daedalus framework provides a complete end-to-end design flow. However, tool interoperability in this framework was found to be a significant problem that required a great deal of software engineering effort, and the need for industrially relevant case studies was noted [98].

3.6 Summary of existing design tools

An aim of this chapter was to determine whether any existing tool can solve the problems listed at the end of Chapter 2. These problems were **IP integration**, **system-level DSE**, **multiple platforms** and **heterogeneous communications**. In this chapter, various tools were presented that address these problems, but the discussion of these tools has raised additional problems that must be addressed. These include:

High-level component design the ability for new components to be designed as an integral part of the flow from a high-level specification;

QoR control the ability for designers to extract high quality of results by specifying precisely the implementation of a component;

Interoperability the use of standards-based languages, and the ability for users to modify the flow by, for example, writing custom DRCs and optimisations.

Many of the tools discussed in this chapter provide some of these features, and a selection of these tools will now be summarised and compared to determine whether any of them satisfy all of the requirements.

CAL is a high-level language for the description of dataflow actors, providing back-ends which allow the generation of implementation code for both software and hardware platforms.

MCF is a metamodeling framework which demonstrates the use of XML metadata for libraries of components to assist in the task of system-level design space exploration.

SPW is a signal processing system design tool that allows graphical composition of processing blocks and the creation of new blocks using a proprietary C interface. However, these blocks cannot be converted automatically to FPGA cores.

Ptolemy is a dataflow system modelling tool which focuses on the interactions between components with differing models of computation.

SCF is a component composition framework that deals specifically with the problem

of mapping components to processing platforms in a heterogeneous system and inferring the necessary communications infrastructure.

Daedalus is a suite of tools which allow the generation of hardware components from MATLAB or SANLP specifications, the importing of existing library components, the exploration of different mappings to execution platforms, and the inference of communications infrastructure.

VIPI (Vivado IP Integrator) is a tool that allows pre-existing IP cores to be connected together, with automatic DSE performed through propagation of core parameters and interoperability resulting from the use of standard IP-XACT component and design descriptions.

VHLS (Vivado HLS) is a high-level synthesis tool allowing C code to be converted into FPGA components.

MaxCompiler is a tool that converts input descriptions in a Java-based language to hardware components. Control over quality of results is possible through the use of Java API calls to customise the generated code and integration with existing software is supported.

System Generator is a tool that allows the composition of DSP IP blocks, and is a predecessor of VIPI.

From these descriptions, it can be inferred that each tool has certain strengths, and these are summarised in Table 3.2 with a tick to indicate a well-supported feature of a tool, and a cross otherwise.

Broadly, it can be deduced from this table that the tools that are most appropriate for the design of high-performance components (CAL, VHLS, MaxCompiler) typically offer limited support for the modelling and analysis of systems comprising a large number of components, and conversely, those that focus on system design (MCF, SCF, VIPI, System Generator) often do not offer a component description input. Where attempts have been made to provide both of these features (SPW, Daedalus), interoperability is typically lacking and users are locked into the design methodology and constraints of the tool. For

Table 3.2: Feature matrix for a selection of existing design automation tools.

	CAL [89]	MCF [69]	SPW [88]	Ptolemy [86]	SCF [72]	Daedalus [94]	VIPI [75]	VHLS [24]	MaxCompiler [54]	System Generator [73]
IP integration	×	✓	✓	×	✓	✓	✓	×	×	✓
System-level DSE	×	✓	✓	✓	×	✓	✓	×	×	✓
Multiple platforms	✓	×	×	×	✓	✓	×	✓	✓	×
Heterogeneous comms	×	×	×	×	✓	✓	×	×	✓	×
High-level components	✓	×	✓	×	×	✓	×	✓	✓	×
QoR control	×	×	×	×	✓	×	×	×	✓	×
Interoperability	×	×	×	×	×	×	✓	×	×	×

^a The C and Java input languages could be regarded as standard, but the compiler directives and compiler modifications are non-standard and neither tool outputs a metadata description for generated components which would allow simpler integration in tools such as IP Integrator.

these reasons, it can be concluded that none of these tools solve all of the problems listed in Chapter 2 without significant modifications.

3.7 Conclusion

This chapter describes a number of additional limitations of existing design tools beyond those presented in Section 2.3, and these include high-level component design, control over quality of results and interoperability. In searching for a single tool which addresses these limitations, it is determined that none of the existing tools fully address all of the limitations described. Thus, a novel tool flow is required which combines existing tools, where possible, with newly-developed infrastructure that adds the missing features. There are a number of ways to develop an improved toolflow, and one that has been developed in the course of my EngD research will be outlined in the next chapter.

Chapter 4

Architecture

Chapter 2 described the problems encountered in the design of FPGA systems and Chapter 3 described the approaches taken to solve similar problems in the academic literature, concluding that no existing tool is sufficient to solve all of the problems described. This chapter begins with the description of an idealised *design process* demonstrating the evolution of a system design from specification to implementation, and then continues with the description of a *tool flow* that enables this process through a coordinated sequence of tool invocations. The chapter concludes with a summary of how the features of the flow address the problems listed at the end of Chapter 2.

4.1 Idealised design process

An idealised design process is shown in Figure 4.1 and the steps are explained in the following text.

- 1. Component instantiation** IP blocks are instantiated where available to fulfil functional requirements of the system, and abstract components without an associated functional description are created and instantiated where IP blocks are not available. The IP blocks are provided with metadata describing their interface data types and token rates, but this information is left unspecified in custom blocks at this stage.
- 2. Coordination** System topology is defined through connection of components, with

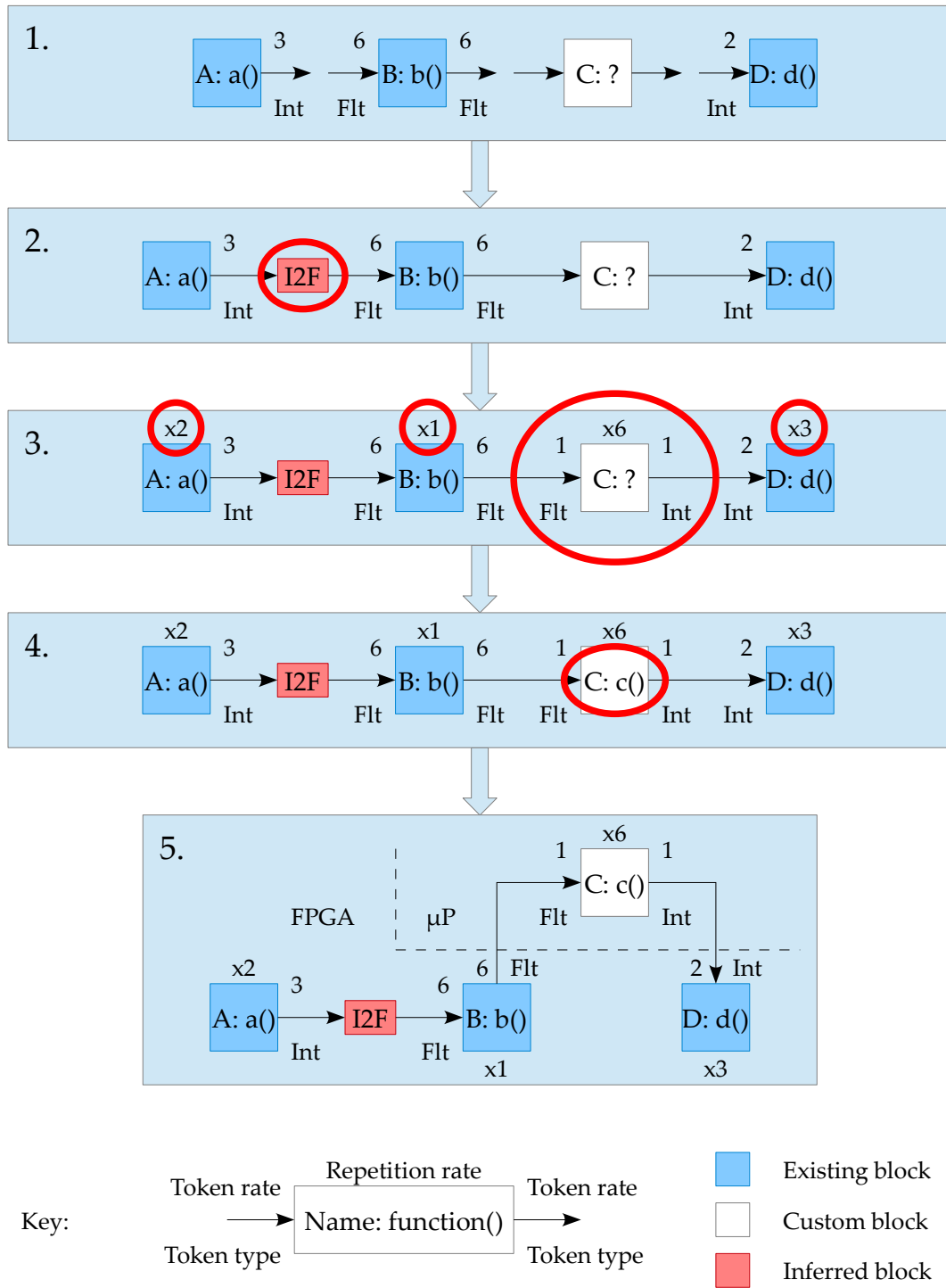


Figure 4.1: An idealised design process that addresses the issues raised in Chapter 2. Some of the new characteristics introduced in each stage are circled.

DRCs being run based on available metadata. Where DRCs fail in a way that allows an automatic resolution, blocks are inserted to handle the incompatibilities that were identified; where this is impossible, blocks must be inserted manually.

3. **High-level DSE** Using the IP metadata, the data rates throughout the system are determined automatically and candidate data types on the interfaces of the custom blocks are proposed. Where a number of solutions are possible, the designer selects the appropriate option. For example, in Figure 4.1, block C could process one token at a time and repeat this six times in a rate-matched system, or could process two tokens, repeating three times.
4. **Functional correctness** Components are refined by adding software implementation detail that defines the functional operation of the components: software models of cores, or custom software code for custom components. The design is executed and profiled to identify performance bottlenecks.
5. **Hardware mapping** Software models of cores are replaced with the cores themselves, with hardware-software communications infrastructure instantiated automatically between the cores and the remaining software components. Where software components form a bottleneck, they are re-implemented in HDL (this could be done automatically using high-level synthesis). The system is then profiled again, and further software-to-hardware migrations of components are tested until performance targets are met.

4.2 Tool flow overview

A tool flow architecture supporting this design process is outlined in Figure 4.2. The important features of this flow are as follows:

- an intermediate representation (IR), which stores dataflow, data type and system coordination information;
- high-level inputs allowing this IR to be generated;

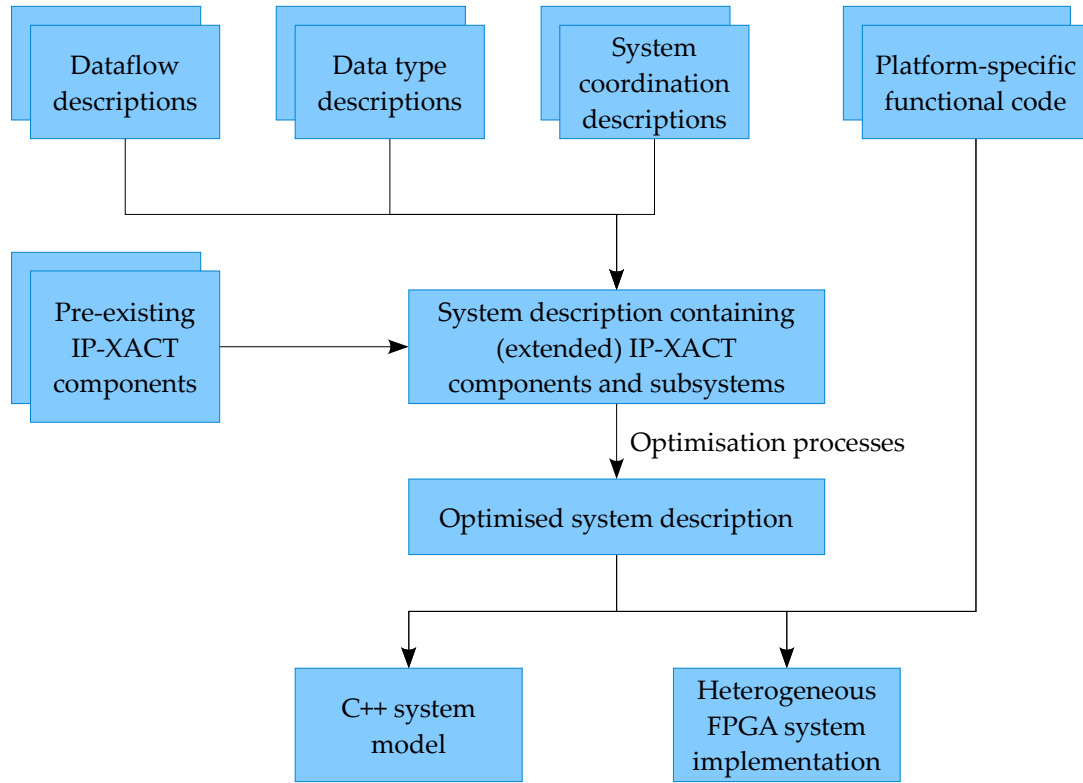


Figure 4.2: Tool flow overview.

- system-level optimisation processes which make use of the information exposed in the IR to improve system performance and/or resource usage;
- back-end code generation processes for a variety of platforms which incorporate platform-specific functional code that is provided by the user into a system framework that is generated automatically.

Each of these features is described in further detail in the following sections.

4.2.1 Intermediate representation

An important concept in system design is orthogonalisation of concerns [37], as discussed in Chapter 3. Currently, these concerns are represented in C++ and HDL, with significant overlap, and in a way that prevents the extraction of individual concerns: for example, it is difficult to determine whether a core has an AXI interface from automatic analysis of the HDL code, even though the reverse process of generating an AXI interface in HDL is relatively simple. The overlap of concerns in C++ and HDL arises from the representation

of platform-independent concerns in platform-specific languages, and one of the goals of this thesis is to show how this can be avoided by representing the platform-independent concerns orthogonally to each other in models conforming to a standardised metamodel. In order to maintain **control over quality of results**, however, the platform-specific concerns are specified in their original platform-specific languages.

From these models, it is possible to produce platform-specific implementations on **multiple platforms** using model-driven engineering: code is generated from the platform-independent concerns and merged with the existing platform-specific code. To avoid the **tool interoperability** issues encountered in the Daedalus toolflow [98], it is desirable to adopt or develop a standard intermediate representation to link the stages in the MDE flow. This approach allows a variety of high-level input languages to be used and allows code for a variety of output platforms to be generated, and has been demonstrated successfully in tools such as LLVM [99].

Since the IP-XACT standard was designed to represent some of the concerns listed in Chapter 2 and may be extended to represent the others, it is suitable for use in this context. The decision to base the tool flow around IP-XACT was made early in the project, before it had been widely adopted in Xilinx, and later adoption of this standard in the organisation vindicated this approach: IP-XACT component descriptions are now defined for cores by core developers and stored in the Vivado IP Catalog, and IP-XACT design descriptions are created for systems in Vivado IP Integrator. Thus, existing components which already have IP-XACT specifications can be integrated and extended with new metadata where necessary, and component composition frameworks that output IP-XACT design descriptions, such as Vivado IP Integrator, may be ‘plugged in’ as a front-end design environment to the tool flow.

In order to improve **IP integration** to the level required for the design process in Figure 4.1, however, the standard IP-XACT metadata descriptions must be extended. The requirements for an improved metamodel are derived from the analysis of a number of existing cores, and two types of metadata extensions are required: data type extensions and dataflow extensions. In contrast to the RTL layer metadata described in Chapter 2, the data type layer and behaviour layer metadata on each core can be difficult to deter-

mine. In most cases, the required information can be extracted by reading data sheets. However, some aspects of these layers are not stated explicitly in data sheets and must be determined through experimentation or through assistance from other users or from the original core developer. In passing, an immediate benefit of metamodeling becomes clear: even aside from the code generation or interoperability benefits, the provision of unambiguous core specifications conforming to a unified metamodel enhances basic usability and thus encourages IP reuse. The data type and behavioural layer metadata is outlined below and described in more detail in Chapter 5.

Data type metadata has two functions: firstly, to determine component interoperability, and secondly, to allow communication between diverse architectures such as microprocessors and FPGA fabric for the purposes of hardware testing and heterogeneous system implementation. An important aspect of this is data reformatting using data type encoding and decoding functions. These functions are normally written manually, but may be generated automatically from the metadata descriptions of data types if these are already defined for the purposes of interoperability.

Dataflow metadata allows components' production and consumption of tokens to be considered as a platform-independent concern, and therefore allows code performing this function to be automatically generated. The justification for a dataflow abstraction arises from the recent adoption of AXI in Xilinx cores: many IP developers have chosen the streaming variant of this standard (AXI4-Stream) to appear on the interfaces. In the Xilinx LTE systems, streaming is ubiquitous, since any remaining memory-mapped interfaces are wrapped with streaming wrappers and FIFO buffers are added to allow latency insensitivity. Thus the LTE systems appear much like dataflow process networks consisting of self-scheduling actors. While the LTE software models conform to a looser formalism, in which code is written manually to coordinate the components (known in Ptolemy as the *Component Interaction* domain), it is demonstrated in Chapter 7 that the coordination code can be generated automatically, and thus the dataflow model is no less applicable in the software models. We propose that if the system is represented at a high level as a dataflow graph, software and hardware components can be interoperable with no need for manual scheduling code.

4.2.2 High-level inputs

As indicated in this chapter, this thesis focuses more on the elaboration of a metamodel for FPGA cores and systems than the design of particular high-level languages with which they may be described. The design of a high-level language is governed firstly by the metamodel and secondly by market-dependent issues which are the subject of ongoing work elsewhere in Xilinx. For example, development of a new user-facing tool for the description of IP metadata was underway during the course of my research, but since this work was not complete, it was not practicable to build on it, and since the work had already begun, there was little benefit in developing an alternative from scratch. Thus, the scope of the challenge in defining high-level inputs to the design flow was limited to solving the immediate challenges of automating the production of verbose and repetitive XML metadata descriptions.

The new Xilinx tool for the description of IP metadata is known as the IP Packager. This is a GUI application that may be controlled using Tcl, and it constructs an IP-XACT definition based on the parameters that are set in the tool. This tool is used to create the metadata files for many of the cores in the Xilinx IP Catalog. While the IP Packager allows simple parameters to be added to IP-XACT components and bus interfaces, it does not permit arbitrary hierarchical XML structures to be added, which is a key requirement of the extensions presented in Chapter 5. For this reason, I have implemented two flows for adding the metadata extensions: the first extends the XML output from the Packager tool with additional information, while the second avoids the IP Packager by creating XML definitions from scratch in the form expected by downstream tools. Both options require a way to describe cores according to the metamodel described in Chapter 5, until this is possible using the IP Packager, and rather than designing new languages that may ultimately be replaced as the IP Packager evolves, a number of existing languages are used and extended where necessary.

To enter dataflow information for cores, the CAL language from the OpenDF project is used, and to describe data types, input mechanisms are implemented from the industry-standard ASN.1 language [100] and a language used internally within Xilinx called RMAP.

To enter system coordination information, which is already supported by IP-XACT, the NL language is used primarily and integration with the Vivado IP Integrator tool is also demonstrated. Input flows from these languages, together with a discussion of how they may be used to generate software simulation models, will be described in further detail in Chapter 7.

4.2.3 Optimisations

A wide variety of low-level optimisations are performed by software compilers and HDL synthesis tools. However, these tools are limited in their ability to perform high-level optimisations that modify the algorithm of the design. Viewing optimisations as endogenous model transformations at a higher level of abstraction has a number of advantages: firstly, it allows optimisation for quantitative design quality metrics such as latency or memory requirements that cannot be extracted from low-level code; secondly, the design space exploration loop is tighter; thirdly, it provides guidance in the implementation of the parts of the system that cannot be automatically generated.

4.2.4 Code generation

Depending on the performance requirements of the system, a variety of different execution platforms must be targeted, possibly consisting of a heterogeneous mix of processors and FPGA fabric. Each of these platforms has a set of platform constraints, as described in Chapter 3 which determine the nature of the code that is executed on that platform. The main execution platforms for Xilinx LTE systems are the XMODEL environment and (currently) the ML605 FPGA development board with a MicroBlaze processor. A number of additional platforms are sometimes used, such as HDL simulation in the ModelSim tool.

Of these platforms, I have targeted the XMODEL framework. The modelling in software of a system described in an IP-XACT representation may be achieved in a number of ways, which can generally be categorised under one of the following two approaches:

Interpretation of the model A software framework may be constructed which reads in

IP-XACT components and designs, instantiates the appropriate components at run-time and executes the composed network of software components;

Generation of implementation code from the model The IP-XACT descriptions may be used to generate software code.

Applying the first approach in the context of the XMODEL framework requires significant changes to the framework. To avoid creating another version, I opted for the latter approach, and the process of generating C++ code for this platform is described in Chapter 7.

Stefan Petko has also worked on generating code for the Xilinx ML605 platform [101], and we have collaborated on a hardware-in-the-loop environment which links C++ modelling and FPGA implementation using IP cores. To address the requirement for **heterogeneous communications**, Stefan is using the IP-XACT representation of the system to generate communications infrastructure¹.

4.3 Implementation aspects

The tool flow is constructed as a series of model transformations between XML documents. Input specifications are transformed into XML as soon as possible, such that the majority of the transformations are performed in the XML domain. Rather than having a single, large transformation from input to output, the flow is comprised of a pipeline of transformations of limited scope. There are a number of advantages to this XML pipeline arrangement:

- Since the input and output to each stage is an XML document, the flow may be extended without great difficulty by inserting additional processing stages. Transformations may also be reordered where necessary.
- There is a large ecosystem of technologies and tools surrounding XML, such as XPath, XSLT and XML Schema. These allow the efficient querying, transformation

¹This will be presented in his EngD thesis, which at the time of writing is work in progress.

and validation of XML documents representing the incremental refinements of the design.

- The scope of the captured data in each stage of the flow can easily be extended, without modification of low-level parsers and data structures. Component characteristics that are specific to a particular model of computation may be layered on top of metadata describing concepts that are germane to a number of models, and the choice of DRC and optimisation procedures that are applied may be governed by the presence or otherwise of the metadata required as their inputs. For example, if token production and consumption rates are present in the metadata, then dynamic dataflow scheduling code may be generated. If these rates are also static, then synchronous dataflow may be applied to generate static schedules.
- Debugging is simple since the output of each transformation stage is an XML document that is (relatively) human-readable.

Many of the transformations in the flow are described as Extensible Stylesheet Language (XSL) Transformations (XSLT). An XSL *stylesheet* is an XML document consisting of a number of templates, with each template specifying how XML elements matching a particular pattern should be replaced or augmented with another XML sub-tree. These stylesheets are applied by an XSLT *processor* which examines an input document and transforms any nodes which match a template: thus, no manual traversal of XML documents must be specified in the stylesheet. Since XSL stylesheets are specified using XML, the syntax is quite verbose, but the underlying language concepts are simple.

To be usable in a design flow, the model transformations must be coordinated somehow, and this may be done using technologies such as XProc [102] or Make [103]. XProc is a recent W3C Recommendation comprising an XML schema for describing the composition of XSL transforms, and could be used to coordinate the transformations in model-driven engineering. However, it can only accept XML input, and another mechanism is required to parse high-level languages into XML which cannot be coordinated by XProc. Instead, I use Make to coordinate the model transformations. An advantage of Make is that it is demand-driven, and will not regenerate intermediate files unnecessarily in a

tool flow. However, a problem with Make in the context of MDE is that it coordinates coarse-grained invocations of OS-level processes, and this imposes an efficiency cost. For a multi-stage XSLT-based MDE flow, this requires an XSLT processor to be invoked and terminated repeatedly for each of the required transformations, and this cost is increased when using a Java-based XSLT processor due to the need to bring up the Java Virtual Machine on each invocation. To avoid this overhead, the XSLT processor is run as a Linux “daemon process” using a tool called NailGun [104], and messages containing transformation requests are passed to this process. With this approach, Make only needs to coordinate the execution of small front-end processes and thus the total processing time is significantly reduced.

4.4 Conclusion

Figure 4.3 summarises the architecture of the tool flow, showing the input languages used, whether the contribution at each stage is a novel model, metamodel or both, and the integration with Stefan Petko’s work. From the diagram, it can be seen that C++ and HDL components are generated from a combination of the IP-XACT component descriptions and from manually-specified functional code, and that these are combined into systems using IP-XACT design descriptions. Selective mapping of components to platforms is not addressed but would be a good target for future work.

Referring back to the problems described in Sections 2.3 and 3.6, the tool flow outlined in this chapter addresses these problems in the following ways:

IP integration The metadata that is associated with cores is extended with data type and dataflow information as described in Chapter 5 in order to allowing more effective design rule checking. In principle, automatic coercion is also possible once expressive metadata is in place. One aspect of this, in the context of high-level array data types, is demonstrated in Chapter 6.

System-level DSE By using a model-driven engineering approach, design tradeoffs may be evaluated before the full system is generated, increasing the number of turns per















	Model	Metamodel	Transformation
Pre-existing			
Defined by me			
Defined by Stefan Petko			
Future work			

Figure 4.3: Tool flow architecture, showing contributions.

day (TPD). Once the characteristics of existing IP are specified explicitly in extended metadata, this information may be propagated throughout the system to constrain the implementation choices in the remaining custom components in the system.

Multiple platforms Code for multiple platforms can be generated from a common representation of the platform-independent concerns of the system. Generation of software modelling code is presented in Chapter 7.

High-level component design The CAL, ASN.1 and RMAP languages are used to define high-level component characteristics in a concise manner, and to produce a platform-independent model describing those characteristics as described in Chapter 7. These languages are selected on the basis of their applicability to the problem domain and their extensibility and adaptability, and they are somewhat cumbersome when used in combination. In future work, it would be desirable for these languages to converge, or for their characteristics to be combined in a more elegant manner.

QoR control To allow high performance in generated systems, the platform-specific (i.e. low-level functional) concerns are specified in platform-specific languages separately from the platform-independent models.

Interoperability The toolflow is based around the IP-XACT standard, allowing the importing of existing IP components that are provided with IP-XACT descriptions. It also uses IP-XACT design descriptions to describe the hierarchical structure of the components in the system, allowing compatibility with existing CCFs. At the backend, rather than creating a new software modelling framework, it was decided that C++ components should be generated such that they work within the XMODEL environment that is used to build LTE simulation models.

Chapter 5

A metamodel for Xilinx IP cores and systems and its representation in extended IP-XACT

Chapter 2 presented a number of issues that could be solved if additional metadata were provided for IP cores. RTL layer metadata is already provided in IP-XACT component descriptions, but data type layer and behaviour layer metadata is not defined and proposed solutions in schemas such as XDF and CHREC XML are not sufficiently expressive to capture complete descriptions of Xilinx IP core behaviour. The purpose of this chapter is to specify a metamodel in which this additional metadata can be represented, and to allow it to be more widely shared, understood and integrated, it is represented in an XML schema that is layered on top of IP-XACT. The extended metadata is demonstrated in the context of a selection of cores in the Xilinx IP Catalog.

Although the metamodel was implemented in practice as an XML schema, it will be described in this chapter using an equivalent format that is more concise. Rather than creating a custom pseudo-code language for this purpose, the YAML language [105] will be used. Like XML, YAML can be used to associate data values with hierarchically structured tags, but YAML is targeted towards serialisation of data (or metadata) rather than document markup. This means that, for example, closing tags are not required, which

significantly reduces the vertical space requirements of a YAML description (on the left, below) when compared to an XML description (on the right):

<pre>root: sub: subsub: value</pre>	<pre><root> <sub> <subsub>value</subsub> </sub> </root></pre>
---	---

While YAML syntax is fairly self-explanatory, notes will be added where explanation is required. Newly-introduced features that extend IP-XACT, XDF and CHREC XML are shown in YAML listings in red text¹.

5.1 Requirements

Requirements can be derived through analysis of existing cores, and a commonly-used core which serves as a good example is the FIR Compiler. A filter generated using the FIR Compiler v6.3 has up to four AXI4-Stream interfaces, depending on the chosen configuration parameters, and these are named S_AXIS_DATA, M_AXIS_DATA, S_AXIS_CONFIG and S_AXIS_RELOAD, but for the purposes of this discussion these names will be abbreviated as DIN, DOUT, CONFIG and RELOAD. Each of the four interfaces of the FIR receives streams of data elements that are padded to the nearest byte boundary, and the streams are structured in different ways as shown in Figure 5.1.

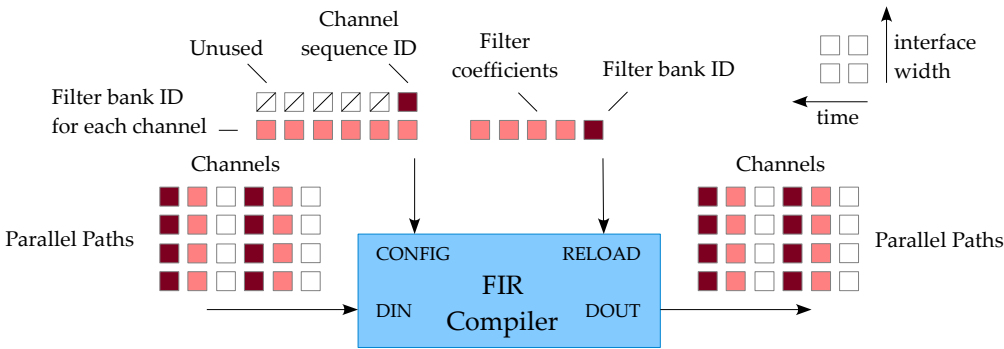


Figure 5.1: FIR Compiler interfaces and data formats.

On the DIN and DOUT interfaces, data samples are communicated as a stream of fixed-

¹In greyscale reproductions, the red text will appear grey.

point values with total width and fractional width specified by core-level parameters, and depending on the configuration of the core, it may operate on multiple independent streams of data. These streams may be interleaved (time-division multiplexed) and/or communicated in parallel with the width of the data interfaces scaled according to the number of parallel streams. In the first case, the streams are known as *channels* (not to be confused with AXI channels), and in the second case, they are known as *parallel paths*.

When the core is configured with more than one channel and more than one set of filter coefficients, the coefficient set (filter bank) used by each channel may be selected at run-time using the CONFIG interface. A channel sequence can also be specified, which determines the relative input and output rates of each channel. When the core is configured to support reloadable filter coefficients, the RELOAD interface allows filter coefficients to be assigned to filter banks within the core, and consumes a filter bank identifier followed by a number of filter coefficients.

Various problems are presented in this description of the FIR. The first problem to be discussed lies in the description of arbitrary-width types. Software languages typically include fixed-width integer and floating-point data types that correspond to the capabilities of typical microprocessors, which operate on values of a fixed word length (for example, 32 or 64 bits). While smaller types (for example, the `short int` type in the C language) are permitted in order to allow memory savings, and may allow greater performance due to increased cache locality, their progression through a processor pipeline is not generally faster and additional instructions may be required to extract the correct sequence of bits from a larger load operation. In FPGA designs, the data type impacts significantly on processing efficiency, since narrower types allow resource savings which can be used to implement additional pipelines or to reduce the size of the required device. So in hardware, there is a richer variety of data types, including integers, fixed-point or floating-point real values, and complex values, each with customisable width and precision. Table 5.1 shows the leaf-level data types for a selection of Xilinx horizontal and wireless DSP cores.

Another problem arises from the variety of methods for encoding data for transmission over streaming or memory-mapped interfaces. This encompasses the position of fields

Table 5.1: Leaf-level data types in Xilinx DSP cores

Core ^a	Number set(s)	Number format(s) ^b
DFT	Complex	$(D, D - 1)$
DDS	Complex or real ^c	$(D, *)$
XFFT	Complex	$(D, *)$ or FP32
FIR	Real	$(D_{in}, F_{in}), (D_{out}, F_{out})^d$
DUC/DDC	Complex or real ^c	$(D_{in}, *), (D_{out}, *)$
Channel Estimator	Real	$(D_{in}, D_{in} - 1), (D_{out}, D_{out} - 1)$
MIMO Decoder	Real	$(16, 15), (32, 15)$
MIMO Encoder	Complex	$(D_{in}, *), (D_{out}, *)$
Channel Decoder	Complex	$(16, 14), (16, 10), (8, 3)$
Channel Encoder	Bit	N/A
PUCCH	Complex	$(16, *)$

^a Names are as in Table 2.1.

^b The notation (a, b) represents total width a and fractional width b . D represents a core parameter specifying total width, and F represents a parameter specifying fractional width. A fractional width of $*$ means that the core operates independently of any particular fractional width value.

^c These cores can operate on complex or real values, depending on a core parameter.

^d In full-precision mode, the output total width and fractional width are set automatically to accommodate bit growth.

in structs and the position of elements in arrays: fields may be packed together or separated by padding, and array elements may be laid out sequentially across data beats or communicated in parallel across an interface, depending on timing and resource constraints. In cores such as the DUC/DDC, data encoding is dependent on configuration parameters, as shown in Figure 5.2.

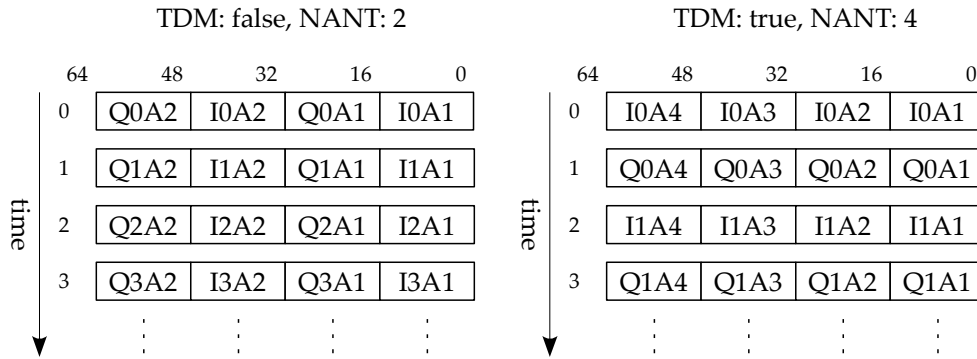


Figure 5.2: DUC/DDC Compiler data format in two modes: no TDM and 2 antennas; TDM and 4 antennas.

Other complexities that must be captured are that fields within structures may be present or not present depending on core parameters, field widths may be parameterisable, and mappings from represented values to encoded values are required to save space in the encoded data format: for example, the PUCCH nant control field which specifies an antenna count of 1, 2 or 4 antennas maps to values of 0, 1 or 2 respectively in order to fit within 2 bits. Furthermore, array dimensions may be present or not present depending on a parameter (such as the ‘channels’ dimension in the FIR data interface arrays, which is only present when multi-channel behaviour is enabled); they may also be fixed-size or variable size, and in contrast to the multidimensional array specifications in languages such as C, it is not just the outer dimension that may be variable. The size of variable-sized dimensions may be specified in a number of different ways, including configuration-time or run-time parameters; in some cases, such as in the LTE DL Channel Encoder v2.1, the total size of the array is not specified at all, with the end of an array being signalled using the AXI LAST signal. Table 5.2 shows the array dimensions for the tokens communicated on the data interfaces of a selection of Xilinx cores².

²In multidimensional streams, the meaning of a token is somewhat ambiguous: at one extreme, the base element could be regarded as the token, and at the other extreme, the sequence of arrays arriving at the interface could be regarded as a top-level, infinite-sized array dimension. In this context, the definition that

Table 5.2: Data interface array dimensionality in Xilinx DSP cores

Core ^a	Data input interface				Data output interface			
	Array dims.	S/P ^b	O ^c	Size ^d	Array dims.	S/P ^b	O ^c	Size ^d
DFT	Elements	S	×	P	Elements	S	×	P
DDS	Channels	S	✓	C	Channels	S	✓	C
XFFT	Elements	S	×	C/P ^e	Elements	S	×	C/P ^e
	Channels	P	✓	C	Channels	P	✓	C
FIR	Packets	S	✓	L	Packets	S	✓	L
	Elements ^f	S	✓	P	Elements ^f	S	✓	P
	Channels	S	✓	C	Channels	S	✓	C
	Paths	P	✓	C	Paths	P	✓	C
DUC/DDC	Carriers	S	✓	C	Carriers	S	✓	C
	Antennas	P	✓	C	Antennas	P	✓	C
Channel Est.	Subcarriers	S	×	F	Codewords	S	×	P
					Subcarriers	S	×	F
MIMO Dec.	Antennas	S	×	U	Codewords	S	×	U
	Subcarriers	S	✓	F	Subcarriers	S	✓	F
MIMO Enc.	Codewords	P	×	F	Antennas	P	×	P
Channel Dec.	Symbols	S	×	P	TB Data	S/P	×	P
	Codewords	S	×	P				
	Subcarriers	S	×	P				
Channel Enc.	TB Data	S	×	L	Encoded data	S	×	L
PUCCH	Symbols	S	×	P	(N/A)			
	Subcarriers	S	×	F				
	Antennas	S	×	P				

^a Names are as in Table 2.1.^b Elements in this dimension communicated in sequential data beats (S) or in parallel across the interface (P).^c Whether this dimension is optional based on core configuration, or not.^d Size specification: fixed (F), core configuration (C), control packet field (P), TUSER channel (U) or TLAST signal (or pre-AXI equivalent) (L).^e FFT transform size may be specified either as a core parameter, or in a control packet.^f An 'Elements' dimension is present when advanced channel sequences are enabled. The size of each element array is different, and determined by the channel pattern which is specified in a CONFIG packet.

5.2 Data type specification

In the Xilinx Video IP group, efforts have already been made towards ensuring consistency. Since video IP uses a more standardised set of data formats than DSP and wireless IP, each format has been associated with an identifier and each of the video cores has a list of supported formats associated with each interface in metadata. The first seven of the available formats are shown in Table 5.3 and the set of these formats that are available on each core interface is shown in Table 5.4.

While this domain-specific approach works when producing systems consisting of cores from single domains, it presents challenges when cross-domain IP connections are required. One common example is the need to connect the RGB output of a video core to a FIR core, with each video channel mapping to a separate channel in the FIR: if an explicit metadata description of the RGB format is not provided, automatic comparisons of the data types on the two interfaces are not possible.

IP-XACT allows simple types like integers and strings to be associated with parameters, and allows register maps to be defined which consist of fields with specified bit widths and bit offsets. However, data types cannot be associated with these fields and streaming packets cannot be defined.

Explicit and platform-independent descriptions of data types may be created using one of a number of *interface description languages* which address the need to standardise data communication between distributed software processes written in different software languages. Examples of these languages include Corba IDL [106], Protocol Buffers [107] and Thrift [108]. However, due to their software orientation, these languages cannot be used to describe hardware-specific features such as variable-width types and type encodings.

These problems are tackled to some extent in ASN.1 [100], which is a mature and wide-ranging standard that was designed to describe the data types sent in communications protocols. ASN.1 deals with data type encodings through sets of *encoding rules*, of which a number are defined including Basic, Canonical, Distinguished and Packed Encoding

will be used is that the token is the array comprising all of the non-infinite dimensions (such as channels) and excluding the infinite dimensions (such as time).

Table 5.3: A selection of data interface formats used in Xilinx video processing cores

Code	Video format	Contents of data word n , $n = \dots$			
		3	2	1	0
0	YUV 4:2:2	—	—	V, U	Y
1	YUV 4:4:4	—	V	U	Y
2	RGB	—	R	B	G
3	YUV 4:2:0	—	—	V, U	Y
4	YUVA 4:2:2	—	α	V, U	Y
5	YUVA 4:4:4	α	V	U	Y
6	RGBA	α	R	B	G

RGB: red, green, blue; A/ α : transparency; Y: luminance; U/V: chrominance.

Table 5.4: Supported data format codes in a selection of Xilinx video cores

Core ^a	Input format	Output format
Color Correction	{1, 2}	{1, 2}
Chroma Resampler	{0, 1, 3}	{0, 1, 3}
Edge Enhancement	{1}	{1}
Gamma Correction	{0, 1, 2, 3}	{0, 1, 2, 3}
Noise Reduction	{1}	{1}
RGB to YCrCb	{1, 2}	{1, 2}
YCrCb to RGB	{1, 2}	{1, 2}
AXI4S to Video Out	Any	N/A
Video In to AXI4S	N/A	Any

^a **Color Correction** : Color Correction Matrix v4.00.a; **Chroma Resampler** : Chroma Resampler v2.00.a; **Edge Enhancement** : Image Edge Enhancement v4.00.a; **Gamma Correction** : Gamma Correction v5.00.a; **Noise Reduction** : Image Noise Reduction v4.00.a; **RGB to YCrCb** : RGB to YCrCb Color-Space Converter v5.00.a; **YCrCb to RGB** : YCrCb to RGB Color-Space Converter v5.00.a; **AXI4S to Video Out**: AXI4-Stream to Video Out v1.0; **Video In to AXI4S** : Video In to AXI4-Stream v1.0.

Rules (BER, CER, DER and PER) [109, 110], and new rulesets can be devised using Encoding Control Notation (ECN) [111]. However, in the case of Xilinx IP, encodings are determined on a case-by-case basis rather than from a ruleset, and using ECN to describe the encoding for each core requires specifications that are overly verbose.

The limitations of existing tools and languages suggest that a novel approach is required. With the representation of core metadata in the extensible IP-XACT format, an opportunity arises to extend this format to include data type information. If data types and their encodings are defined in metadata, functions that encode/decode bus transactions to/from VHDL records or C structs may then be automatically generated, and this will be presented in Chapter 7.

An IP-XACT extension schema will now be described that has been designed to allow data types to be described and then associated either with fields in register-based interfaces or with streaming interfaces, allowing accurate descriptions of streaming data packets to be created. Data types may be basic ‘leaf’ types or hierarchical types such as structures and arrays. The basic types will be discussed first.

5.2.1 Basic type descriptions

Basic types include booleans, integers, reals and complex values. Bit widths are added to data type descriptions as in the example in Listing 5.1, which represents an unsigned data type of (5,4)³. A similar specification can be produced in CHREC XML.

Listing 5.1: Metadata representation of a simple data type.

```
dataType:
  real:
    bitWidth: 5
    signed: false
    fixedPoint:
      fractionalWidth: 4
```

Integers and floating point types are captured in a similar manner, using the integer el-

³In the XML schema, the `signed` and `unsigned` XML tags are used with no data contents, but YAML tags must have data values and so `signed: true` and `signed: false` are used.

ement in place of `real` and the `floatingPoint` element in place of `fixedPoint`. Floating-point values are a generalisation of the IEEE 754-2008 standard to arbitrary total widths and significand widths, as has been implemented in the Xilinx Floating-Point Operator core, with the total width specified using the `bitWidth` element and an additional `significandWidth` element specifying the width of the significand. Floating-point types in this schema are always signed.

Complex values are also permitted which can hold two integer or real values, and the width of the aggregate type is determined by the width of the integer or real subtype. Thus, only a single additional parameter is required for the `complex` element, which specifies whether the real or the imaginary part of the type occurs first (i.e. earlier in the stream, or closer to the least-significant bit in a transaction): `realFirst` or `imaginaryFirst`⁴.

```
dataType:
  complex:
    real:
      bitWidth: 5
      signed: true
      fixedPoint:
        fractionalWidth: 4
    realFirst: true
```

It is desirable to be able to restrict the range of values that may be represented in a field, and IP-XACT allows this to be done for untyped register fields using the `writeValueConstraints` and `enumeratedValues` elements. However, these only constrain the value as represented in hardware, rather than the value in a type, and so with the addition of typed fields, these elements are moved underneath type descriptions. Doing this also helps to address the problems posed by the PUCCH `nant` field, which can be addressed with the addition of an `encodedValue` element under each enumerated value:

⁴These appear as `realFirst: true` and `realFirst: false` in YAML.

```

dataType:
  integer:
    bitWidth: 2
    enumeratedValues:
      - name: ant_1
        value: 1
        encodedValue: 0x0
      - name: ant_2
        value: 2
        encodedValue: 0x1
      - name: ant_4
        value: 4
        encodedValue: 0x2

```

In the absence of value constraints, the permitted values in a data type are determined by its bit width.

5.2.2 Hierarchical composition of types

Leaf types may be composed into a hierarchy of types consisting of structures and multi-dimensional arrays. Structures contain `field` elements which have a base type associated with them, and these base types contain `bitOffset` elements that specify the encoding of the fields in the structure:

```

dataType:
  structure:
    field:
      name: first
      bitOffset: ...
      dataType:
        ...
        bitWidth: ...
    field:
      name: second
    ...

```

A similar approach is taken in the base IP-XACT schema in the specification of register maps. The main difference is that a `structure` is a more abstract entity than a register map, which with the addition of bit widths and bit offsets, can represent either a register map or a streaming packet. Other differences are that the `field` here is an abstract field which is only assigned an encoding upon the addition of bit width and offset information

to lower-level subtypes, and that the `bitOffset` element stores the offset from the start of the structure rather than from the start of a word. Under this approach, a separation of concerns is enforced between the width of the physical channel or register (at the RTL layer) and the width of the data type (at the data type layer) transmitted across it.

Arrays are also representable in the schema, and these may be multidimensional, with each array dimension having a name such as “channels”, “antennas” or “subcarriers”. Each dimension can be marked as being optional, and can have a `size`. The presence of optional dimensions and the specification of configurable array sizes are discussed later in this chapter.

```
dataType:
  array:
    name: antennas
    size: 4
    dataType:
      array:
        name: subcarriers
        size: 12
        dataType:
          ...
```

5.2.3 Data type encoding

It is necessary in the schema to describe both abstract types and their encodings. While ASN.1 enforces a separation between the abstract type and its encoding, these aspects are rarely considered in isolation from each other in the context of the data formats on FPGA cores. The approach proposed here is less rigid in separating these concerns: while types can be specified independently of a register map or streaming packet implementation context, the internal layout of the encoded type in a stream or register map is intermingled with the abstract type description.

In IP-XACT, the locations of registers and register fields are described using bit offsets in a one-dimensional address space, and mapped storage locations are specified using the `bitOffset` element. In streaming packets, there are two address dimensions to consider: space (across the width of a bus interface) and time (across multiple transactions on that

interface), as demonstrated in the DUC/DDC data encoding example in Figure 5.2.

When considering streaming data arrays such as those used by the DUC/DDC, the mapping of the elements in an n -dimensional array to a 2-dimensional space becomes complicated. A similar problem has previously been tackled in the Array-OL language [71]: in this language, the mapped location vector⁵ \mathbf{e}_i of each element \mathbf{x}_i of an input array is specified by a $2 \times n$ fitting matrix, F , and an origin vector \mathbf{o} such that $\mathbf{e}_i = \mathbf{o} + F \cdot \mathbf{x}_i$. This equation models array encoding essentially as an affine transform from array indices to two-dimensional storage locations, and this requires two values to be specified for each input array dimension.

This approach could be used to map multidimensional array elements to locations in time and space in transactions on a bus interface, but a simpler specification can be used if a policy is adopted such that bit offsets increase across interface widths from least-significant bit to most-significant bit and then across transactions in time order, meaning that the encoded address space is linear. Mapping to a one-dimensional stream means that a $1 \times n$ fitting matrix may be used, with a single value for each dimension. These values are then essentially *strides*, as used in Fortran 90 and MPI [112], and specifying a single stride value for each array dimension (or complex value) allows the position of each element in time and space to be specified precisely. Figure 5.3 demonstrates how this may be done in practice.

5.2.4 Naming and reference

So far, types have been described independently of the lower-level channel over which they are to be communicated. It is assumed here that streaming types are associated with individual IP-XACT ports rather than bus interfaces since a number of ports may comprise an interface and each of these may have a unique data type (for example, the DATA and USER channels comprising an AXI4-Stream interface).

One implication of associating types with ports is that there are two bit widths to be considered: the width of the port, and the width of the type that is communicated over

⁵Notation has been altered for clarity.

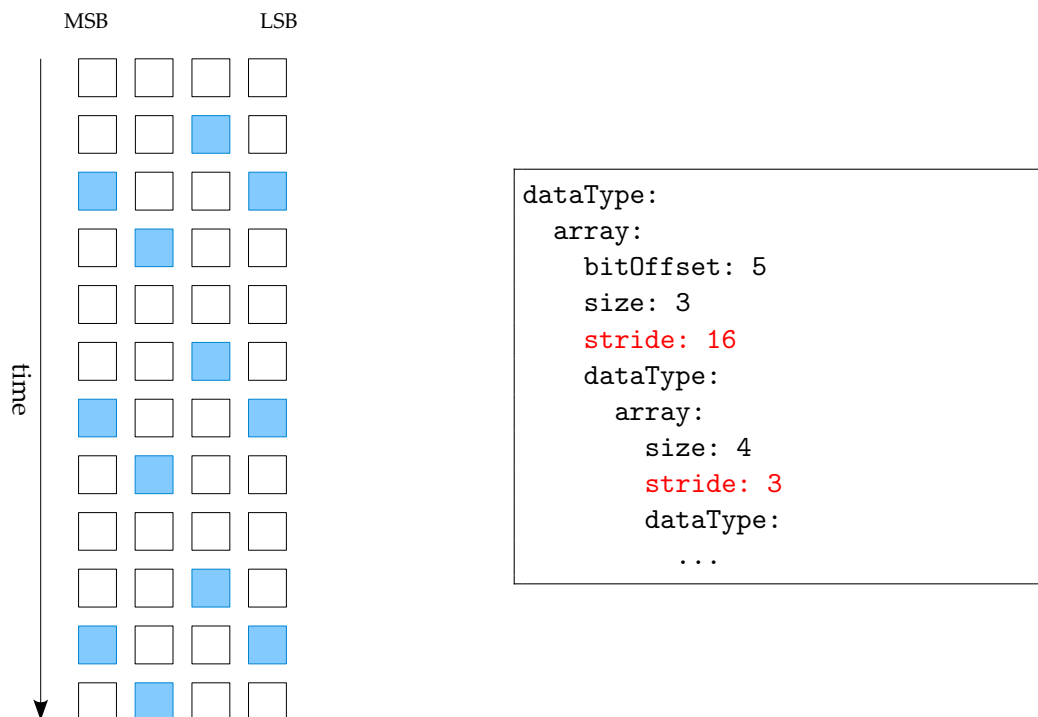


Figure 5.3: Metadata representation of complex array strides. On the left is a diagram showing a sequence of transactions on an interface in a regular pattern. The pattern can be represented as an array with two dimensions that starts at the 5th byte in the stream, with each of the three repetitions of the outer dimension occurring 16 bytes apart and each of the four repetitions of the inner dimension occurring 3 bytes apart.

that port. This is useful when sending, for example, a 5-bit signed value over an 8-bit AXI4-Stream interface, since the width of the type can be used to determine that the sign should only be extended to the 5th bit rather than to the 8th bit.

A data type may be associated inline with an IP-XACT port as follows:

```
port:
  vendorExtensions:
    dataType:
    ...
```

CHREC XML also allows types to be defined in a component's `vendorExtensions` and referenced throughout that component, but since data types are often shared between components, as in the case of the video IP described earlier, it is also desirable to permit libraries of types to be created, stored externally, and referenced in multiple IP-XACT component descriptions. In IP-XACT, components and designs are referenced using four attributes: vendor, name, library and version (VLNV). These attributes are added to data type descriptions using a `dataTypeDef` field and this is stored in a separate XML file which will be called a "data type description". Named types can then be referenced using a `dataTypeRef` element, as shown below.

```
dataTypeDef:
  vendor: xilinx.com
  library: ip
  name: xfft_ctrl
  version: 8.0
  dataType:
    structure:
    ...
```

```
component:
  ports:
    - vendorExtensions:
        dataTypeRef:
          vendor: xilinx.com
          library: ip
          name: xfft_ctrl
          version: 8.0
```

5.2.5 Parameterisation and dependencies

Values in IP-XACT metadata files are often variable, and may depend on another value stored within the same file. For example, IP-XACT components have a list of parameters, and each parameter has a default value. An example of an IP-XACT dependency that was defined using the XPath `id()` function was shown in Listing 3.2, and in YAML syntax, a suitable way to represent this is to use the `*` and `&` symbols as shown in Listing 5.2:

&ovflo is an anchor for the node immediately following it, and *ovflo is an alias node which references the anchored node.

Listing 5.2: Abstract metadata representation of optional port.

```
component:
  parameters:
    - name: ovflo
      value: &ovflo false
  busInterfaces:
    - name: M_AXIS_STATUS
      vendorExtensions:
        busInterfaceInfo:
          enablement:
            presence: optional
            isEnabled: *ovflo
```

A similar approach is used in the data type schema to specify dependent aspects of data types: for example, this is required when the size of an array is dependent on the value of a configuration parameter or when an array dimension is optional, as shown in Listing 5.3. Note that in this example, default parameter values (4 and false) are specified, and that the presence and isEnabled elements are adopted from the Xilinx vendor extension schema to describe the optionality of array dimensions.

Listing 5.3: Metadata representation of parameterised array dimension size and presence.

```

component:
  parameters:
    - name: nant
      value: &nant 4
    - name: groupsc
      value: &groupsc false
  ports:
    - name: din
  vendorExtensions:
    dataType:
      array:
        name: antennas
        size: *nant
      dataType:
        array:
          name: subcarriers
          size: 12
          presence: optional
          isEnabled: *groupsc
        dataType:
          ...

```

When types are defined externally in a `dataTypeDef`, this form of dependency specification cannot be used, since the scope of an XPath expression is limited to the current document. To avoid this problem, types may be given a number of parameter elements, and the default value of each parameter may be defined at the location of the type reference.

```

dataTypeDef:
  vendor: xilinx.com
  library: ip
  name: xfft_ctrl
  version: 8.0
  parameters:
    - name: has_nfft
  dataType:
    structure:
      field:
        name: nfft
        presence: optional
        isEnabled: *has_nfft

```

```

component:
  parameters:
    - name: C_HAS_NFFT
      value: &C_HAS_NFFT true
  ports:
    - vendorExtensions:
        dataTypeRef:
          vendor: xilinx.com
          library: ip
          name: xfft_ctrl
          version: 8.0
          withParams:
            - name: has_nfft
              value: *C_HAS_NFFT

```

This mechanism may be extended to support parametric type polymorphism, allowing

components such as data sources and sinks to be configured to read or write data of a particular type. The listing below makes use of the XPath `split` function to enable this.

```
component:
  name: data_source
  parameters:
    - name: data_type
      value: &data_type ::xfft_ctrl_packet:1.0
  ports:
    - vendorExtensions:
        dataTypeRef:
          vendor: xilinx.com
          library: ip
          name: split(*data_type, ':')[3]
          version: split(*data_type, ':')[4]
```

Finally, it should be noted that metadata parameters such as array sizes may change dynamically. Capturing the dynamic dependency of one value, such as an array size, on another metadata element is useful when generating executable code from an IP-XACT description, and thus will be described in Chapter 7.

5.2.6 Full examples

To tie together the concepts presented so far, data type descriptions conforming to the metamodel representing packet descriptions for two Xilinx LogiCOREs will be presented. Firstly, the packet communicated on the CONFIG interface of the FIR, in the presence of multiple filters, patterns and channels, may be described as shown in Listing 5.4 (a more verbose specification would be required to deal with the optional presence of elements when there is only a single filter, pattern or channel). The `pad()`, `ceil()` and `log2()` functions are shorthand for XPath expressions that implement each function.

Listing 5.4: Metadata description of the CONFIG control packet received by FIR cores.

```
dataTypeDef:
  vendor: xilinx.com
  library: ip
  name: fir_compiler_config_packet
  version: 6.3
  dataType:
    parameters:
      - name: num_filters
        value: &num_filters 2
      - name: num_patterns
        value: &num_patterns 2
      - name: num_channels
        value: &num_channels 2
    structure:
      - field:
          name: filter_select
          bitOffset: 0
          dataType:
            array:
              size: *num_channels
              stride: >
                pad(ceil(log2(*num_filters)))
                + pad(ceil(log2(*num_patterns)))
            dataType:
              integer:
                bitWidth: pad(ceil(log2(*num_filters)))
                valueConstraint:
                  min: 0
                  max: *num_filters - 1
      - field:
          name: channel_pattern
          bitOffset: pad(ceil(log2(*num_filters)))
          dataType:
            integer:
              bitWidth: pad(ceil(log2(*num_patterns)))
              valueConstraint:
                min: 0
                max: *num_patterns - 1
```

Secondly, Listing 5.5 shows how the DUC/DDC problem can be represented in metadata using parameterised strides: in parallel mode, the complex value has a stride of one element-width, but in TDM mode the stride is the number of antennas multiplied by the element width; in parallel mode, the antenna array has a stride of two element-widths, but in TDM mode the stride is one element width.

Listing 5.5: DUC/DDC Compiler data format expressed in XML metadata.

```
dataTypeDef:
  vendor: xilinx.com
  library: ip
  name: duc_ddc_compiler_data_packet
  version: 2.0
  dataType:
    parameters:
      - name: tdm
        value: &tdm false
      - name: num_antennas
        value: &num_antennas 1
      - name: data_width
        value: &data_width 32
    array:
      name: antennas
      size: *num_antennas
      stride: if(*tdm) then *data_width
              else *data_width * 2
      dataType:
        complex:
          real:
            bitWidth: *data_width
            realFirst: true
            stride: if(*tdm) then *data_width * *num_antennas
                    else *data_width
```

Finally, some of the limitations of the schema should be mentioned. In the FIR core, data channels may be interleaved either in “basic” or “advanced” sequences. With basic sequences, each data channel is processed sequentially in a repeating cycle. With advanced sequences, channels are processed in a pattern such as “0 0 0 1”, “0 1 0 2” or “0 0 0 0 1 2”, where 0, 1 and 2 identify three separate channels. There are 174 possible channel sequences, each specified explicitly with a unique identifier, and they are selected using a field provided over the CONFIG interface.

Advanced channel sequences are difficult to represent in the schema for two reasons: firstly, the data arrays communicated on each channel may be of different sizes, meaning they cannot be represented as multidimensional arrays, and secondly, the stride values are non-uniform. For example, in the pattern “0 0 0 1 0 0 0 1”, the stride in channel 0 varies between 1 (for the first three elements) and 2 (for the fourth element). Therefore, further work is required to determine whether it is possible to represent advanced chan-

nel sequences in an elegant manner.

Additional problems are encountered in attempting to represent variable-sized fields in structures, such as IPv4 options, since the offset of each field cannot be specified in advance. Further work is required to determine how these issues should be handled.

5.3 Component behaviour specification

In this section, metadata extensions are defined which describe the interactions between component interfaces, allowing the data production rate on the component's outputs to be determined when the token consumption on the input ports matches particular, well-defined patterns. By specifying the dynamic behaviour of cores in metadata, the model of computation can be determined from that metadata. So, rather than stating that a core belongs to a particular dataflow domain such as SDF or DDF, the domain emerges from its behavioural properties.

It could be argued that behavioural information is part of the computational concerns of a component, which according to the architecture set out in Chapter 4 is represented in this toolflow in platform-specific languages such as C++ and HDL rather than metadata. However, to the extent that the behaviour is common to multiple platform-specific instances of the component and has an impact on the system-level interconnection of a block, it can be considered as a cross-cutting concern that can legitimately be specified in metadata.

5.3.1 Rate relationships

The first core to be examined is the DFT v3.1. Its ports are already defined in a Xilinx IP-XACT component description, and since it is a pre-AXI core, the ports are not aggregated into higher-level bus interfaces. To define the rate relationships between these ports, the XDF actions element, containing inputs and outputs each with a tokenCount, may be adopted with the proviso that it must distinguish references to IP-XACT ports from references to IP-XACT bus interfaces, each of which may conceptually represent an abstract

dataflow port.

The DFT core has a single action which consumes tokens on all of the input ports and produces tokens on all of the output ports, as represented in the metadata below⁶.

```
component:
  name: dft
  ...
  ports:
    - &xn_re
      name: xn_re
      vendorExtensions:
        dataType:
          ...
    ...
  actions:
    - inputs:
      - portRef: *xn_re
        tokenCount: 1
      - portRef: *xn_im
        tokenCount: 1
      - portRef: *fd_in
        tokenCount: 1
      - portRef: *size
        tokenCount: 1
      - portRef: *forward
        tokenCount: 1
      outputs:
        - portRef: *xk_re
          tokenCount: 1
        - portRef: *xk_im
          tokenCount: 1
        - portRef: *rffd
          tokenCount: 1
        - portRef: *blk_exp
          tokenCount: 1
        - portRef: *fd_out
          tokenCount: 1
        - portRef: *data_valid
          tokenCount: 1
```

In components with a single action linking each of the ports, and with each port associated with a data type, it may be possible to calculate the data rates on each port. In the case of the DFT, the sizes of the input and output data arrays are variable and determined

⁶It is assumed that the clock, clock enable and clear ports belong to a layer below the dataflow representation of the core.

by a transform size that cannot be determined statically, but it can at least be determined from the metadata that control data must be provided for every data array.

Even though the sizes of the data arrays are variable, their variability is constrained by the value transmitted on the `SIZE` port. Capturing this relationship in metadata is not useful in static analysis, but is of use when generating executable code from the metamodel and an approach for capturing this information will now be described.

5.3.2 Dynamic data dependencies

There are a number of characteristics of components that vary dynamically. One is the sizing of variable-sized input and output arrays, and another is action guards, which will be described in more detail in Chapter 7.

Dynamic characteristics of components are a fundamental feature of the CAL dataflow language and XDF IR, and may be represented within the scope of a single action by declaring a variable to be associated with an input token and referencing that variable, or some function thereof, elsewhere in the body of the action definition. It is desirable to adapt this mechanism for use in IP-XACT vendor extensions, making use of existing IP-XACT concepts where possible. To do this, a `declaration` element is added to an action input, as in XDF, and it is referenced in the array size:

```
ports:
- &xn_re
  name: xn_re
  vendorExtensions:
    dataType:
      array:
        name: antennas
        size: *size_value
        dataType:
          ...
inputs:
- portRef: *size
  tokenCount: 1
  declaration: &size_value size_value
  ...
```

This adapted mechanism is only suitable when referencing data on ports that transmit

basic data types such as integers, however, since no mechanism is defined that allows fields in structures or elements in arrays to be referenced.

A comment must be made on the implementation of this mechanism in XML. Elements containing variable data are assigned a `spirit:resolve` attribute, and the value of this attribute determines the resolution mechanism: `user` to indicate configuration by the user and `dependent` to indicate a dependency on another metadata element. To address dynamic dependencies, an additional runtime-dependent value is introduced and an XPath function which extracts the value of a token received on an input port is declared.

```
<spirit:value  
  spirit:resolve="runtime-dependent"  
  spirit:dependency="id('size_value')">0</spirit:value>
```

5.3.3 Timing constraints

An accurate specification of the DFT core must also take account of the constraints on the relative timings of the port transactions. These timing constraints cannot be represented in CAL or CHREC XML, but are necessary in complete descriptions of Xilinx cores such as the DFT. This core requires control tokens on ports such as `SIZE` to be provided on the first beat of each data transaction, which can be specified by adding a `timingConstraint` to the appropriate action inputs. This states that the start of the transaction on the `SIZE` port must occur exactly zero cycles after the start of the transaction on the `XN_RE` interface:

```
...  
- portRef: *size  
  timingConstraint:  
    referencePort: *xn_re  
    referenceBeat: first  
    minLatency: 0  
    maxLatency: 0  
...
```

In contrast to the DFT core, the LTE Channel Estimator v1.1 aggregates ports as bus interfaces and does not permit concurrent streaming of control on the `CTRL` interface and data on the `Y` interface. Since there is no buffering on either interface, control and data must

be communicated sequentially with the data on the Y port following the data on the CTRL port. However, some latency is allowable between these transactions. This scenario may be modelled with a timing dependency of the Y port on the CTRL port and a `bufferDepth` element set to zero on both interfaces⁷:

```
actions:
- inputs:
  - busRef: *AXI4Stream_MASTER_s_axis_y
    tokenCount: 1
    timingConstraint:
      referenceBus: *AXI4Stream_MASTER_s_axis_ctrl
      referenceBeat: last
      minLatency: 0
      bufferDepth: 0
  - busRef: *AXI4Stream_MASTER_s_axis_ctrl
    tokenCount: 1
    bufferDepth: 0
outputs:
  - busRef: *AXI4Stream_MASTER_m_axis_h
    tokenCount: 1
```

In contrast, the LTE PUCCH Receiver retains the constraint that control (on the CTRL interface) must be read before data (on the DIN interface), but since it provides buffering for up to two tokens on each interface, control for the next action may be consumed concurrently with the data for the current action. To model this scenario, the `bufferDepth` element would be set to 2.

In the system design process, it is desirable to know the maximum data throughput of a core, and this can be determined from the rate at which the cores consumes data, known as the initiation interval or introduction interval (II). Comparing the Channel Estimator and PUCCH, the throughput of the PUCCH is greater as a result of the buffering that it provides. To describe the II, CHREC XML provides a `dataIntroductionInterval` element [48], but in latency-tolerant cores supporting standards such as AXI4-Stream, each core can more accurately be said to have a minimum initiation interval (MII) associated with each action. MII values are sometimes provided in core data sheets, but rather than specifying the MII value explicitly in metadata, it may be derived from other informa-

⁷The MASTER component of the interface names is assumed to have been entered in error in the core packaging process.

tion. A bus interface can only receive a single token at a time, so the MII for an action is dependent on the number of transactions required for a token to pass over an interface, which will be referred to here as the “length” of the token. For example, an action with a single input interface that receives a data token over three clock cycles could be said to have an MII of three plus whatever additional delay is required to calculate the output, which will be referred to as the *repetition delay*. Token lengths are determined by their size and the width of the interface over which they are sent, both of which may already be included in the metadata for each bus interface.

For an action with more than one input interface, each of which is unbuffered, the MII for that action is equal to the length of the longest chain of tokens on dependent input interfaces, plus the repetition delay. For example, in the Channel Encoder, the MII is the length of the CTRL token plus the length of the Y token, plus the repetition delay. On the other hand, an action consisting entirely of buffered input interfaces has an MII equal to the length of the largest input token, plus the repetition delay. For example, in the PUCCH, the size of the data token on the DIN interface dominates that of the CTRL interface, so the MII is the length of the DIN token, plus the repetition delay. In general, with mixed buffered and unbuffered input interfaces, the MII for an action is the maximum of the length of the largest input token and the length of the largest chain of unbuffered dependent input interfaces, plus the repetition delay.

In summary, the MII can be calculated rather than being specified explicitly, and thus, in contrast with CHREC XML, a repetition delay element (*repeatDelay*) is defined in this schema for each action, allowing the MII to be derived for each action separately. If the *repeatDelay* element is not present, it can be assumed to be zero.

One final aspect of core timing is output latency. The latency of configured Xilinx cores is currently determined through one of a number of methods:

- provision of latency examples for certain core configurations in the data sheet; or
 - on-the-fly calculation during core configuration in the Vivado design environment;
- or
- provision of a formula in the data sheet.

In the absence of any of these specifications, manual experimentation is required to determine the latency of a core. In order to ensure that latency is specified wherever possible in a way that may be understood by a design environment, it is desirable that a flexible mechanism for latency specification is provided. CHREC XML specifies latency information using an integer in a `pipelineDepth` element, but a more flexible and precise and specification can be achieved by associating worst-case and/or best-case token production latencies on particular outputs with the arrival of data on particular inputs. In the DDS core, the latency of the core is configurable and can be specified in metadata as follows:

```
actions:
- inputs:
  - busRef: *S_AXIS_PHASE
  ...
  outputs:
  - busRef: *M_AXIS_DATA
  ...
  timingConstraint:
    referenceBus: *S_AXIS_PHASE
    referenceBeat: last
    maxLatency: PARAM_VALUE.LATENCY
```

5.3.4 Blocking

In the FIR Compiler LogiCORE, there is an additional notion of *blocking* relationships between input interfaces. In the DFT, which is a pre-AXI core, the `FD_IN` port signals the readiness or otherwise of all of the input ports. AXI4-Stream interfaces, in contrast, provide a `VALID` signal allowing such tests to be made on a per-interface basis.

In the FIR, if a `CONFIG` token arrives but no `DIN` tokens arrive, the `CONFIG` token is not processed. Similarly, if a `RELOAD` token arrives but no `CONFIG` token arrives, the `RELOAD` token is not processed. The `CONFIG` channel is said to block on the `DIN` channel, and the `RELOAD` channel is said to block on the `CONFIG` channel. To allow correct, automatic, integration of the FIR, these characteristics must be captured in the metamodel.

This behaviour cannot be represented in CHREC XML, but it can be described using dataflow actions. The FIR may be modelled using three actions: one for ordinary pro-

cessing of DIN tokens, one to select a new filter bank (through the CONFIG channel), and one to update filter banks (through the RELOAD channel). The impact of each action upon the core's state is also represented with the introduction of state elements, adapted from XDF, together with new action inputs and outputs that indicate explicit manipulation of the component's state. The only part of the total state that is represented in metadata is that which has an effect on the system-level behaviour of the core, and state ports do not require a token count, since repeated reads or writes are idempotent.

The first action processes a DIN token into a DOUT token, using the configuration information that is held in the core's state.

```
state:
- &filter_bank_data filter_bank_data
- &active_filter_bank active_filter_bank
action: &action_1
  name: action_1
  inputs:
    - busRef: *din
      tokenCount: 1
    - state: *filter_bank_data
    - state: *active_filter_bank
  outputs:
    - busRef: *dout
      tokenCount: 1
```

The second action describes the selection of a filter bank in the core using the CONFIG channel. Since the DIN channel blocks on the CONFIG channel, a DIN token is also included in the list of inputs. The CONFIG token can arrive up to two cycles after the DIN token in order for the updated configuration to take effect, and this can be represented in metadata using a timing constraint.

```

action: &action_2
name: action_2
inputs:
  - busRef: *din
    tokenCount: 1
    timingConstraint:
      referenceBus: *config
      referenceBeat: last
      minLatency: 2
  - busRef: *config
    tokenCount: 1
  - state: *filter_bank_data
outputs:
  - busRef: *dout
    tokenCount: 1
  - state: *active_filter_bank

```

The third action describes the reloading of filter coefficients. It takes a variable number of tokens from the RELOAD interface, a single token from the CONFIG interface, and a single token from the DIN interface, and writes the result to internal state, as represented in the listing below. Timing constraints for this action are unspecified in the data sheet, and note that since a variable number of RELOAD packets may be processed in a single CONFIG update, no tokenCount is provided for the RELOAD input.

```

action: &action_3
name: action_3
inputs:
  - busRef: *din
    tokenCount: 1
  - busRef: *config
    tokenCount: 1
  - busRef: *reload
outputs:
  - busRef: *dout
    tokenCount: 1
  - state: *filter_bank_data

```

These actions must be prioritised such that action 3 will fire preferentially to action 2, which will fire preferentially to action 1. The concept of action priorities is defined in CAL, and is required to model accurately the following characteristics of the FIR: firstly, that if a DIN token arrives, it will terminate a configuration update if a CONFIG token is available, and secondly, that if a CONFIG token arrives, it will update the filter banks if

RELOAD tokens are available.

```
component:
  actionPriority:
    - *action_3
    - *action_2
    - *action_1
```

Listing action priorities is not sufficient to ensure correct operation when the latency of the input channels is variable, however: with the adoption of the AXI4-Stream standard with variable-latency interconnect blocks and the desire for IP cores to participate in heterogeneous GPP-FPGA systems, the correct relative timing of the token arrivals cannot be preserved. For example, to pass data from software memory to an instantiated FIR core in FPGA fabric, the following code sequence may be desirable:

```
fir_instance.reload_write(reload_token);
fir_instance.config_write(config_token);
fir_instance.din_write(din_token);
```

Since the interconnect latency is variable, these tokens may arrive out-of-order. While this is not a problem in fully-blocking cores, it may lead to nondeterministic execution in cores such as the FIR. Cores that exhibit this problem can be identified with a simple analysis of their metadata: since there are multiple actions, and since the same state variable is listed as the input to at least one action and as the output of at least one other, there is a data dependency between them. Thus, there is a race condition between the actions, but the metadata allows this fact to be determined easily in software.

To ensure deterministic behaviour, a number of approaches may be taken. One option is to send all FIR data through a single channel in which data order can be preserved, but this limits the data throughput and requires additional channel deaggregation logic to be instantiated in the FPGA fabric. A similar approach would be to associate a sequence number with each channel transaction. Perhaps the most preferable option would be for all cores to be implemented with a “dataflow” mode, in which race conditions between input interfaces are not possible. The selection of the most appropriate approach is beyond the scope of this document, but in the process of defining the metadata, it has at least been pointed out that the problem exists and that a decision must be made.

Table 5.5: Dynamic interface behaviour of Xilinx DSP cores

Core ^a	Rate	Input timing	Blocking	Buffered interfaces
DFT	Static	Coincident	None	None
DDS	Variable	Control before data	None	All
XFFT	Variable	Dependent on signal ^c	None	S_AXIS_DATA, 16 elements
FIR	Variable	Control before data	Partial ^b	All
DUC/DDC	Variable	Control before data	None	Unspecified
Channel Est.	Static	Sequential	Full	None
MIMO Dec.	Static	No dependency	Full	None
MIMO Enc.	Static	Coincident	None	None
Channel Dec.	Static	Control before data	Full	External ^d
Channel Enc.	Variable	Control before data	None	Unspecified
PUCCH	Variable	Control before data	Full	All, 2 elements

^a Names are as in Table 2.1.

^b RELOAD blocks on CONFIG, CONFIG blocks on DIN, otherwise non-blocking.

^c It is only safe to send new control data to the core when its `event_frame_started` output signal is asserted.

^d Input codewords are buffered in an external memory accessed through the core's M_AXI interface.

5.3.5 Summary

In this section, the dynamic behaviour of a number of Xilinx DSP cores was examined. A characterisation of their behaviour is summarised in Table 5.5, and the key observations from this exercise are that the following characteristics must be represented in metadata:

- relative rate of token consumption and production, which may be static or variable;
- timing constraints between the arrival of data tokens on input interfaces: in some cases, tokens must be coincident, while in others they must arrive sequentially or in a specified order, and in others, there is no timing dependency;
- blocking dependencies between interfaces: some interfaces may block whilst another interface is waiting for data;
- depth of buffers on interfaces, if present.

Thus, it was proposed that specifications of the cores' dynamic behaviour can be built by deriving actor-oriented abstractions with timing annotations, and it was shown how these specifications can be layered on top of IP-XACT descriptions.

5.4 Discussion

While general trends in IP design can be observed in the examples provided in this chapter, it can also be noted that many cores in the library differ from the standard form in some way. The reasons for these differences vary from differing system design assumptions, resource constraints, stage of interface standardisation process, customisability requirements, and even the particular design philosophies adopted by different groups of engineers in an organisation. For example, full blocking of data and control tends not to be implemented on horizontal cores because of the additional resource cost that this imposes – this functionality is regarded as unnecessary by customers implementing systems with predictable inter-block latency, and thus the resource cost takes precedence in the decisions taken during the implementation of these cores. On the other hand, blocking is regarded as essential in baseband LTE systems because of the unpredictability of control data arrival times from an embedded microprocessor, and so the LTE baseband cores tend to implement full blocking behaviour.

Different cores have different use cases, and must therefore be designed in different ways. However, there is some value in attempting to eliminate unnecessary heterogeneity in a library of IP, and thus some proposals are presented here which would address this concern.

- Horizontal cores could have two modes that may be selected at configuration-time: synchronous reactive, and dataflow. The latter would imply full blocking amongst all interfaces and allow integration into variable-latency scenarios such as LTE or “accelerator” systems.
- A policy of either providing sufficient buffering within cores to allow some latency-insensitivity, or a policy of not providing this buffering and making latency con-

straints explicit in data sheets, or a mixed approach in which both buffering capacity and latency constraints are made explicit.

- Core latency could be stored as XPath expressions in metadata, allowing both the generation of latency estimates at core configuration time and the use of this information in a design environment to capture (and potentially optimise) total system latency.

5.5 Conclusion

This chapter has described a metamodel for Xilinx IP cores, and has demonstrated how various cores conform to that metamodel. The metamodel is implemented in XML, and extends the IP-XACT, XDF and CHREC XML schemas with contributions including:

- structure and array data types that can be used to describe streaming control and data packets;
- complex types;
- strided array types allowing the description of sparse arrays, or of arrays that interleave data from multiple dimensions;
- flexible encodings of enumerated types;
- external data type definitions that may be referenced by multiple components;
- adaptation of dynamic dependencies as found in XDF for use in IP-XACT vendor extensions;
- the use of XDF actions to describe data rates on component interfaces;
- timing constraints and latency characterisation between component ports and interfaces; and
- representation of core blocking characteristics using dataflow actions that interact via state.

In evaluating the utility of this metamodel, the primary consideration I will use is whether it saves time by allowing code to be generated. Other considerations such as the succinctness of the metadata specifications in comparison to other possible metamodels are believed to be subjective to some degree, and are difficult to evaluate outside of peer review⁸.

The benefits of the metamodel in code generation are demonstrated in Chapters 6 and 7 for a number of the metamodel features including descriptions of structured data types and the adaptation of XDF dataflow specifications for use in an IP-XACT-based schema. However, a number of limitations of the schema were pointed out, including the difficulty of representing complex data structures such as IPv4 options and advanced channel sequences in the FIR Compiler. Additionally, it remains to be proven that the use of timing, latency and blocking metadata in the proposed format can be used to generate wrappers for cores which allow them to be integrated automatically into dataflow systems. Such a demonstration would be a useful target for future work.

In the next chapter, it will be shown how the array metadata proposed in this chapter can be used to optimise buffering in multidimensional signal processing systems.

⁸While positive feedback was received from the reviewers of my conference papers, an attempt at standardisation would be a more thorough test. When I proposed the metadata extensions to the IP-XACT technical committee, their response was that they had not seen widespread demand for these types of extensions, and thus proceeding with a standardisation effort would not be prioritised at that time.

Chapter 6

Tool-assisted design of multidimensional streaming systems

In Chapter 2, it was stated that the LTE systems consist of a sequence of blocks that operate sequentially over different data dimensions, that reorder buffers were introduced between the blocks, and that the positions of these blocks had an impact on system-level latency and memory requirements.

The purpose of this chapter is to demonstrate firstly that metadata describing the structure of array data types, as presented in Chapter 5, can be used to automatically infer appropriate data reordering blocks, and secondly, that automated techniques may be used to infer these blocks in the most efficient positions in the data stream. While this process is applicable to any system that operates on multidimensional data, it will be applied in this chapter to the LTE Uplink Receive system that was shown in Figure 2.1.

The structure of the chapter is as follows. First, it will be shown how the need for buffers can be determined automatically from metadata, and then, some options for the implementation of these buffers will be presented. Then it will be shown that while a variety of buffering “scenarios” are possible in a system, each can have different latency and memory requirements, and an example of this issue will be presented. The remainder of the chapter then shows how efficient buffering scenarios can be determined automatically, and it will be demonstrated how this process can be used to construct an efficient LTE

system from a combination of pre-existing and custom components.

A patent describing this work has been issued by the US Patent and Trademark Office with the title “Method for determining efficient buffering for multi-dimensional datastream applications”, and a copy of this document is included in Appendix A.

6.1 Determining buffering requirements

The need for buffers between components is influenced by the array data types on their interfaces, which may be encoded in metadata in the format proposed in Chapter 5 with dimension names abbreviated as specified in Table 5.2. A list of the array dimensions that are present in the data transmitted or received over a component interface may then be extracted using concise specifications in the XPath language. For example, the channel matrix input to the MIMO Decoder is a two-dimensional array of antennas and code-words, and when it is expressed as follows:

```
<x:dataType>
  <x:array>
    <x:name>ant</x:name>
    <x:dataType>
      <x:array>
        <x:name>cw</x:name>
        ...
      </x:array>
    </x:dataType>
  </x:array>
</x:dataType>
```

the list of dimensions can be extracted with a single line of XPath code:

```
string-join(descendant::x:array/x:name, ', ')
```

which generates the string ant, cw.

In the following discussion, the list of dimensions present in an array type on a component interface will be referred to as the *interface dimension list* (IDL), $\mathbf{i} = (i_1, i_2, \dots, i_n)$, where i_1 is the outer dimension and i_n is the inner dimension. Using this notation, the

MIMO Decoder output has an IDL of (cw), and since the IDFT to which it is connected in the LTE Uplink Receive system operates over arrays of subcarriers, its input interface may be given an IDL of (sc).

The importance of these IDLs is that they play a part in determining the need for reorder buffers between components, such as that shown in Figure 2.4, but the need for these buffers is not determined through direct comparison of the dimension lists on the interfaces of the cores. For example, between a core that outputs (A,B) and another core that inputs (B), no reorder buffer is required because repeated firings of the downstream block mean that it consumes the A dimension implicitly; we shall say that A is added to the *repetition dimension list* (RDL), \mathbf{r} of the downstream component. Instead, it is the *effective* IDL (which will be called the EIDL) on each interface that determines the need for a buffer, where $\mathbf{e} = \mathbf{r} \parallel \mathbf{i}$, i.e. the IDL prepended with the contents of the RDL: $(r_1, r_2, \dots, r_m, r_1, i_2, \dots, i_n)$. A fuller discussion of how the components' RDLs are determined in general will be provided later in this section.

A buffer is required between two interfaces with unequal EIDLs, and this buffer must store all of the dimensions that occur below any corresponding pair of dimensions that are unequal. In the previous example with an output list of (A,B) and an input list of (B), the effective dimension list of both components is (A,B), so no buffer is required. Between the lists (A,B,C,D) and (A,C,B,D), however, the dimensions that must be stored and reordered are B, C and D, but not A. The total memory requirement of a typical reorder buffer is the product of the sizes of all the dimensions that are stored: in the latter example, $|B| \times |C| \times |D|$.

Returning to the LTE example, the MIMO Decoder must fire $|\text{sc}|$ times, where $|\text{sc}|$ is the size of the 'sc' dimension. This effectively transforms the dimension list on the output interface from (cw) to (sc,cw). Similarly, the IDFT must fire $|\text{cw}|$ times, such that its input interface effectively has a dimension list of (cw,sc). In the process detailed above, the need for a subcarrier dimension in the IDFT has been propagated to the MIMO Decoder and the need for a codeword dimension in the MIMO Decoder has been propagated to the IDFT, as shown in Figure 6.1.

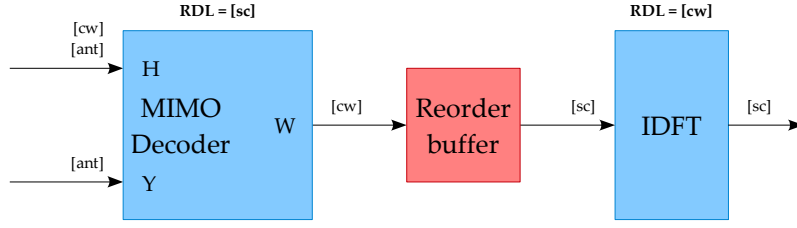


Figure 6.1: Propagation of dimensions.

6.2 Automatic buffer instantiation

With knowledge of the dimension lists on the interfaces of the MIMO Decoder and IDFT and the repetition lists for each component, the generation and instantiation of this reorder buffer and the calculation of its memory requirements can be performed automatically. In the example in Figure 6.1, the reorder buffer must store all of the codewords and all of the subcarriers. The buffer that performs this task is essentially an N -dimensional interleaver or deinterleaver, where N is the number of dimensions to be buffered. In one possible implementation, input data is stored in memory in the order it is received, and then read out in a strided fashion: the address of the j^{th} element, A_j , in the output stream is calculated from the output array type as follows:

$$A_j = \sum_{p=1}^N \left[(j \bmod |i_p|) \cdot \prod_{q=p+1}^N |i_q| \right]$$

where $|i_p|$ and $|i_q|$ are the sizes of the p^{th} and q^{th} dimension of the output array type, counting from the most-significant to the least-significant, and N is the number of dimensions as before. In an alternative implementation, data is stored in a strided fashion and read out in order, while for high throughput scenarios, a non-blocking ping-pong buffer may be used.

In the case of 2D transforms, the existing Xilinx Symbol Interleaver/Deinterleaver (SID) LogiCORE, appropriately configured, may be used to reorder array dimensions. This approach is also suitable for higher-dimensional transforms that are reducible to a 2D transform: an example is (A, B, C) to (C, A, B) , since A and B may be amalgamated into a single dimension. However, (A, B, C) to (C, B, A) cannot be reduced in this manner.

For higher-dimensional transforms that cannot be reduced to a 2D transform, there are at least two alternative solutions. One is to instantiate a chain of SID cores, and the other is to generate a custom reorder buffer component.

6.2.1 Chaining SID cores

Using a chain of two instances of the SID core, (A, B, C) could be transformed to (C, B, A) via an intermediate stage of (A, C, B) . For any pair of multidimensional arrays, it is possible to determine a minimal chain by considering whether there are any common sub-lists in both of the dimension lists, and aggregating the dimensions in those sub-lists. For example, transforming (A, B, C, D, E) to (D, E, B, C, A) may be done by viewing B and C as a new dimension P , and D and E as Q , and then transforming (A, P, Q) to (Q, P, A) .

The disadvantage of the SID-chaining approach is that it will lead to greater memory requirements and greater latency than if a reorder buffer is implemented from scratch. Between (A, B, C) and (C, B, A) , a custom block would buffer $|A| \times |B| \times |C|$, while the chain of SID instances proposed earlier would buffer $|B| \times |C|$ for the first SID and $|A| \times |B| \times |C|$ for the second SID. The increase in memory is bounded in general, however: the top dimension need only be moved once, the second dimension need only be moved once, and so on. In the worst case, each dimension has a cardinality of 2 and the cost of a single custom buffer is 2^N , where N is the number of dimensions. The cost of a sequence of 2D SIDs is $2^N + 2^{N-1} + \dots + 2^3 + 2^2$. The limit of this sequence is approximately 2^{N+1} , so the cost of a SID chain is, at most, twice that of a custom buffer. In many real-world cases, however, the memory penalty of this approach will be lower and the advantages offered by IP reuse could outweigh the small memory advantage of a custom implementation. For example, if $|A|$ is 12, $|B|$ is 7 and $|C|$ is 4, the cost of two SIDs between (A, B, C) and (C, B, A) would be either 420 or 364 depending on whether the smaller SID swaps the A and B dimensions or the B and C dimensions, versus 336 for a custom reorder buffer implementation. Thus, in this case, the overhead of the SID implementation is between 8.3% and 25%.

6.2.2 Custom implementation

If custom implementation is necessary, the implementation task may be simplified with the aid of high-level synthesis, which has been shown to give high-quality results for simple loop nests such as those that are required in a reorder buffer. Since these buffers have a regular structure, C code to perform the required transformation can be generated automatically from XML data types using XSLT, and then passed through an HLS tool to generate a custom reorder buffer. Example code for the case of reordering the dimensions in a single LTE resource block with symbol (sym), subcarrier (sc) and antenna (ant) dimensions from (sym, sc, ant) to (ant, sc, sym) is shown in Listing 6.1.

Listing 6.1: Reorder buffer described as C code for input to Vivado HLS.

```
#define MAX_ANT 4
#define MAX_SC 12
#define MAX_SYM 7

void reorder_buffer(const ap_uint<32> in[MAX_SYM] [MAX_SC] [MAX_ANT] ,
                   ap_uint<32> out[MAX_ANT] [MAX_SC] [MAX_SYM] )
{
    #pragma AP array_stream variable=in
    #pragma AP array_stream variable=out

    #pragma AP interface ap_fifo port=in
    #pragma AP interface ap_fifo port=out

    int ant, sc, sym;

    ap_uint<32> buf[MAX_SYM] [MAX_SC] [MAX_ANT] ;

    for (sym = 0; sym < MAX_SYM; sym++)
        for (sc = 0; sc < MAX_SC; sc++)
            for (ant = 0; ant < MAX_ANT; ant++)
                buf[sym][sc][ant] = in[sym][sc][ant];

    for (ant = 0; ant < MAX_ANT; ant++)
        for (sc = 0; sc < MAX_SC; sc++)
            for (sym = 0; sym < MAX_SYM; sym++)
                out[ant][sc][sym] = buf[sym][sc][ant];
}
```

This processes 32-bit values, and VHLS compiler directives (`#pragma AP`) request that the data arrays are passed in as streams (`array_stream`) through interfaces with ready-valid

handshakes (`interface ap_fifo`). Synthesizing this code into HDL using the Vivado HLS tool produces the results in Table 6.1.

Table 6.1: High-level synthesis results for simple reorder buffer.

Latency	LUT usage	Flip-flop usage	Block RAM usage
1296	157	55	1

It is notable in these results that the latency is greater than the expected value of 672, which is the product of the sizes of the dimensions multiplied by two: once for input, once for output. This is firstly because each buffer read takes two cycles, and secondly because each inner loop requires two additional cycles, giving a total of $((4 + 2) * 12 + 2) * 7$ (518) for the input loop nest and $((7 * 2 + 2) * 12 + 2) * 4$ (776) for the output. These problems can be addressed by adding the `pipeline` directive to each loop nest, which allows an initiation interval of 1 cycle on memory accesses and also flattens the nests, with a one-off penalty of 2 cycles. With these modifications, the resource utilisation of the block is increased significantly, as shown in Table 6.2, but the total latency is reduced to 676 cycles, which is close to what could be achieved through hand-coding. Whether or not the additional resources in the modified block justify the reduction in latency is dependent on the requirements of the particular system in which it is to be used, but the advantage of implementing the block in Vivado HLS is that it provides an easy way to switch between the implementation options should this be necessary.

Table 6.2: High-level synthesis results for optimised reorder buffer.

Latency	LUT usage	Flip-flop usage	Block RAM usage
678	336	95	1

A number of further modifications could be attempted in future work. One is for the sizes of the array dimensions to be variable at run-time: for example, the number of symbols depends on whether or not the OFDM cyclic prefix is extended or not, and this is determined by a field in a control packet in the Xilinx LTE systems. Another development would be to implement ping-pong buffering, allowing buffer reads and writes to occur simultaneously. However, since these developments are tangential to the main thrust of

this chapter, they are not discussed here in further detail.

6.3 Determining repetition lists using pairwise propagation

Instantiating buffers automatically requires EIDLs to be determined, which in turn require RDLs to be determined, so a method to determine RDLs must be found. Initially, it should be considered whether this problem can be considered as a special case of data type propagation as implemented in tools such as Simulink and Vivado integrator, i.e. pairwise propagation between interfaces.

In Figure 6.1, two connected interfaces are considered, and any dimensions present in one but not the other are propagated across the connection. The system is now extended by connecting the LTE Uplink Channel Decoder, which has a data type on its input interface of (sym, sc), to the output of the IDFT¹.

Continuing the propagation process described above, the Channel Decoder is integrated by propagating the (sym) dimension to the IDFT. The behaviour of the IDFT (iterating over codewords) has already been specified by the MIMO Decoder, and this is now refined such that it additionally iterates over data symbols. The type propagation from the Channel Decoder to the IDFT may occur either before or after the propagation from the MIMO Decoder to the IDFT. If the propagation from the MIMO Decoder is applied first, as in Figure 6.2, an additional reorder buffer is required when propagating from the Channel Decoder.

Alternatively, when the data type from the Channel Decoder is propagated first as shown in Figure 6.3, no reorder buffer is required between the IDFT and the Channel Decoder, meaning that fewer reorder buffers are required and the total memory requirement is reduced.

The relationship between RDS permutations and system efficiency is reminiscent of a

¹The LTE Uplink Channel Decoder v3.0 User Guide specifies (sym, cw, sc), but since the operations performed on each codeword are independent, the (cw) dimension may be considered as part of the RDL rather than the IDL. Depending on the IDLs of the neighbouring components, removal of dimensions from an IDL can allow more efficient buffering scenarios to be determined.

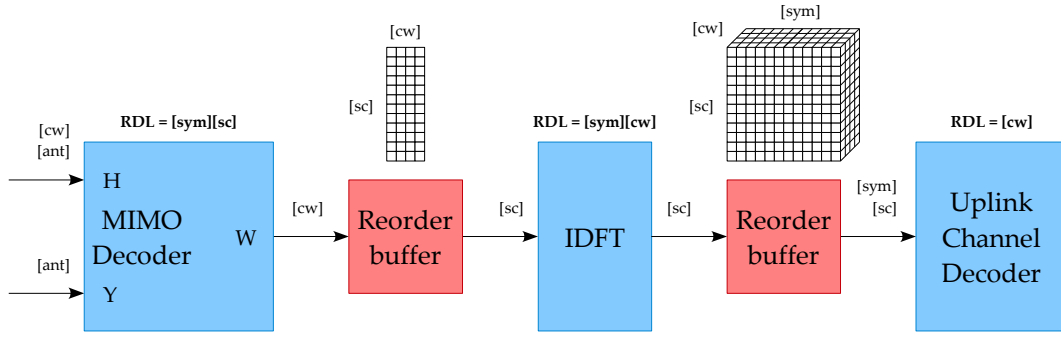


Figure 6.2: Addition of Uplink Channel Decoder block.

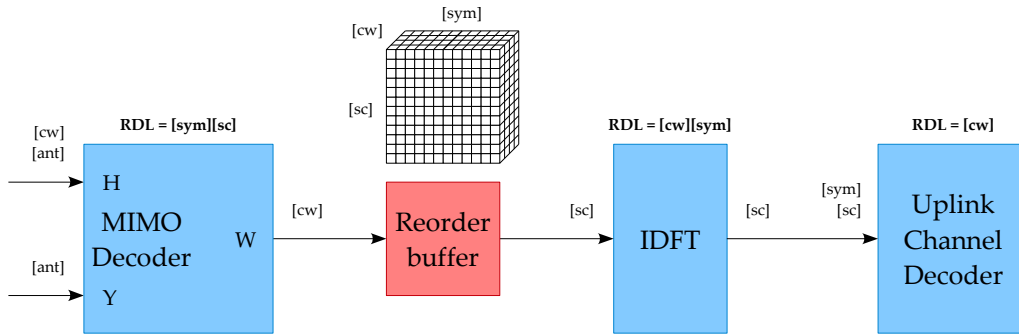


Figure 6.3: Alternative propagation.

common technique in software compilers known as *loop interchange*, in which the order of iteration variables in loop nests is altered. The aims of this technique are different, however: loop interchange is applied in order to maximise locality of reference by aligning array accesses with cache lines.

In this example, two propagation scenarios have been considered and the latter option is superior due to its lower buffering requirements. In this example, the difference is minor and an exhaustive analysis of the available scenarios may be done manually, but the number of scenarios grows very quickly when the numbers of components, c , interconnections, e , and dimensions, d , are increased. The approach described previously requires each interconnection in the system to be examined, and for each interconnection, any dimensions in one interface that are not present in the other are propagated. Some interconnections must be considered more than once, such as in Figure 6.2 where the (sym) dimension is propagated to the MIMO decoder. In the worst case, only a single dimension is propagated on each inspection of an interconnection, meaning that an

interconnection may need to be examined d times, and thus the number of inspections required is $O(de)$. These inspections may be performed in any order, and thus the number of possible propagation scenarios is $O((de)!)$. As a result, this process cannot be applied in systems with large numbers of components and dimensions, and furthermore, it is not known whether it will always find an optimal solution. For these reasons, an approach based on pairwise propagation is not suitable.

6.4 Determining repetition lists using Synchronous Dataflow

As an alternative, it is possible to construct an approach based on Lee's theory of Synchronous Dataflow (SDF) [113]. One of the primary applications of this theory is to determine a firing rate for each component in a system which leads to matched data rates, but here it will be used in an alternative context in which it is used to determine efficient repetition lists. The following discussion will briefly introduce the typical usage of SDF, and will then demonstrate the enhanced application.

First, a topology matrix, Γ , is constructed with a row for each interconnection and a column for each component, in which the elements of the matrix represent the rate that a component produces tokens on an interconnection. In the system in Figure 6.2, the first component (the MIMO Decoder) produces $|cw|$ tokens on the first interconnection while the second component (the IDFT) consumes $|sc|$ tokens on the same interconnection. Similar reasoning may be applied to the second interconnection, and the following topology matrix may be constructed:

$$\begin{bmatrix} cw & -sc & 0 \\ 0 & sc & -sym \times sc \end{bmatrix}$$

A condition for the system to have a set of firing rates that leads to matched data rates is that the topology matrix must have a rank of $N - 1$. As a result of the rank-nullity theorem, the nullspace of such a matrix has a single basis vector of integers, and according to the SDF theory, this nullspace vector represents the repetition vector \mathbf{q} , i.e. \mathbf{q} such that

$\Gamma \mathbf{q} = 0$. This vector can be found using methods such as Gaussian elimination, and using this approach in the case of the system in Figure 6.2 requires the following equation to be solved:

$$\begin{bmatrix} cw & -sc & 0 \\ 0 & sc & -sym \times sc \end{bmatrix} \cdot \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This may be reduced to the following using elementary row operations:

$$\begin{bmatrix} 1 & -sc/cw & 0 \\ 0 & 1 & -sym \end{bmatrix} \cdot \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Since the rank of the matrix is 2, we can introduce one free variable, c , and consider that equivalent to one of the elements of \mathbf{q} . For convenience, we use q_2 . Rearranging the simplified matrix produces the nullspace as follows:

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = c \begin{bmatrix} sc/cw \\ 1 \\ 1/sym \end{bmatrix}$$

An integer nullspace vector is required, but this vector contains fractional elements. Since c is a free variable, it follows that the elements in the vector may be multiplied out to give an integer vector². A minimal such vector (with no replication of dimensions) may be determined by multiplying each element by the least common multiple of the denominators of each of the elements (in this case, $cw \times sym$) to give an integer vector as follows:

$$\begin{bmatrix} sym \times sc \\ sym \times cw \\ cw \end{bmatrix}$$

²This is allowable because the nullspace is determined from the linear combination $a_1v_1 + a_2v_2 + \dots + a_nv_n$ of nullspace vectors v_i and scalars a_i with the scalar values being free.

Thus for matched data rates in the system in Figure 6.2, the MIMO Decoder must fire $sym \times sc$ times, the IDFT must fire $sym \times cw$ times, and the Channel Decoder must fire cw times.

Note that each element of the repetition vector, \mathbf{q} , is an integer formed from the product of the cardinalities of each of the members of a set of data dimensions. Murthy and Lee describe an extension of SDF in which the elements of the repetition vector are themselves replaced with vectors, with each element corresponding to a different dimension [85], but in their approach, the significance of the various possible orderings of the dimensions in these vectors is not discussed. Viewing these vectors instead as sets, with a variety of possible dimension orderings that imply different latencies and memory requirements, allows automatic buffer minimisation techniques to be applied. In the following discussion these sets will be referred to as *repetition dimension sets* (RDS).

6.4.1 Determining repetition sets automatically

Determining the repetition vector using standard SDF scheduling techniques provides a vector of integers, and in order to preserve the dimension names in repetition sets, the SDF calculations must be performed symbolically. A number of software packages may be used for this purpose, and I used a library for the Python programming language called “SymPy” [114] since it can interoperate with other Python tools described later in this chapter (although this interoperability is not demonstrated here). The repetition sets may be determined using the code in Listing 6.2³:

³A more robust implementation would ensure that the rank of the input matrix is $N - 1$, and thus that there will be a single vector in the nullspace of the matrix.

Listing 6.2: Python session demonstrating symbolic determination of an SDF rate vector and an RDS.

```
>>> from sympy import *
>>> cw = Symbol('cw')
>>> sc = Symbol('sc')
>>> sym = Symbol('sym')
>>> T = Matrix([[cw, -sc, 0], [0, sc, -sym*sc]])
>>> q = T.nullspace()[0]
>>> print q
[sc*sym/cw
 [      sym]
 [      1]
>>> q *= lcm(map(lambda x: denom(x), q))
>>> print q
[sc*sym]
[cw*sym]
[      cw]
>>> RDS = map(lambda x: x.as_ordered_factors(), q)
>>> print RDS
[[sc, sym], [cw, sym], [cw]]
```

6.4.2 Determining buffer-minimising repetition lists

Having determined an RDS for each component, an ordering of the elements in each RDS that minimises buffering must be found. This is essentially a constraint optimisation problem. Various tools and frameworks are available that may be used to model and solve constraint satisfaction problems, and optimisation problems may be layered on top of these through the incremental addition of successively tighter cost constraints. One form of constraint satisfaction is satisfiability modulo theories (SMT). In contrast to Boolean satisfiability, which is the problem of determining whether the predicates in a boolean formula can be assigned an interpretation such that the formula evaluates to TRUE, SMT is a generalisation of this to a variety of other background theories such as integers and lists [115].

A variety of SMT solvers have been implemented, and the “Z3” SMT solver from Microsoft Research [116] is one of the most mature. To solve the RDS ordering problem, a set of variables and constraints are presented to the Z3 solver, and each time a satisfying interpretation of the input variables is found, a total cost constraint is added that is lower

than the current cost, and this is repeated until an improved solution cannot be found. This approach may be less efficient than using a purpose-built constraint optimiser, since it is not guaranteed that each iteration with a tightened cost constraint makes use of the work done in previous iterations. However, it appears to be an approach that is accepted in the academic literature [117], and correspondence with the creator of the Z3 solver suggests that successive iterations will reuse previous lemmas unless constraints are removed.

If an exhaustive search were used to determine an efficient buffering scenario, the number of scenarios to test would be $O(d!^c)$ (d factorial to the power of c), where d is the number of dimensions and c is the number of nodes in the system. However, since Z3 uses a backtracking search, it is able to prune subtrees in the search space when the cost constraint is exceeded, with the effect that execution time is reduced.

The system of constraints that is used as input to Z3 is constructed using its Python interface, named Z3Py. Using Python to generate constraints programmatically allows problems to be solved that are inexpressible using the background theories of SMT. This issue will be described later in the chapter.

The solver is provided with an IDL for each interface on each component, an RDS for each component and a list of point-to-point connections in the system. The IDL may be determined using the method shown in Section 6.1, the list of components may be generated from IP-XACT design descriptions, and the RDS may be determined using the symbolic variant of SDF described previously⁴. The scenario in Figure 2.3 may be described as follows:

```
comps = {'mimo': { 'idl': {'w': [cw]}, 'rds': [sc]},
        'idft': { 'idl': {'din': [sc]}, 'rds': [cw]}}

connections = [['mimo', 'w'], ['idft', 'din']]
```

In the tool, an RDL is created for each component, consisting of a list of Z3 variables constructed using the `Const()` function⁵ whose size is equal to the number of dimensions

⁴Direct generation of the tool inputs from XML metadata has not been demonstrated, but no additional metadata elements are thought to be required.

⁵In Z3, constants without an interpretation specified by a constraint are effectively variables, while those

present in the RDS. Each of these variables may be assigned an interpretation by the solver that is a member of the ArrayDim enumerated type of subcarriers, antennas and so on.

```
for block in iter(comps):
    comps[block]['rdl'] = [ Const("x_%s_%s" % (block, dim), ArrayDim)
                           for dim in range(len(comps[block]['rds'])) ]
```

Although the names of specific array dimensions are declared explicitly in this approach, it would also be possible to create the ArrayDim enumeration after a list of required dimension names has been determined from the input data. Note also that while the RDL variables are stored in a Python data structure, they are presented to the SMT solver as a flat list of variables declared using the Const() function, rather than as a hierarchical structure. While Z3 does support the theory of lists, it is somewhat cumbersome to use and is unnecessary in this context.

A number of constraints are then defined for each of the variables created in the array. The first set of constraints states that each variable in an RDL must be assigned a dimension that is present in the corresponding RDS, and these constraints are created using the Or() function.

```
for block in iter(comps):
    for listdim in range(len(comps[block]['rdl'])):
        constraints.append(Or([
            comps[block]['rdl'][listdim] == comps[block]['rds'][setdim]
            for setdim in range(len(comps[block]['rds']))
        ]))
```

The second constraint is that each RDL must contain distinct dimensions. Since the input to the Distinct function must be a non-empty list, empty RDLs are filtered out from the input.

with a defined interpretation are constants in a truer sense of the word.


```
constraints.extend([
    Distinct([
        comps[block]['rdl'][dim] for dim in range(len(comps[block]['rdl']))
    ])
    for block in range(len(filter(lambda x: len(x) > 0, comps[block]['rdl']))
])
```

With these constraints in place, a solver object is created and assigned the list of constraints and is asked to determine whether the constraints are satisfiable. If a satisfying interpretation of the Z3 predicates is found, it must be determined whether this interpretation can be improved. This is done by defining a cost function, adding the constraint that the cost must be lower than the current cost, and attempting to find another solution. The cost function is the sum of the buffering costs on each interconnection, and the cost of each buffer is determined using a recursive function which considers the EIDLs for both components, and if the outer dimensions on each list are the same, considers the next dimensions in the lists, and so on, until a pair of different dimensions is seen. Then, the cost of the buffer is the product of the sizes of the remaining dimensions in one of the two lists.

Solving satisfiability problems of this form requires “satisfiability modulo recursive functions” [118], but this is not supported in the Z3 solver. Instead, the recursive Python function shown in Listing 6.3 may be used to generate hierarchical trees of conditional Z3 constraints which specify the total cost of a reorder buffer for various assignments of dimensions to RDL variables.

Listing 6.3: Python function used to generate Z3 cost constraints.

```
def cost(src_eidl, dst_eidl, dim, counting):
    if not counting:
        if dim == 0:
            return 0
        else:
            return If(src_eidl[dim - 1] == dst_eidl[dim - 1],
                      cost(src_eidl, dst_eidl, dim - 1, False),
                      cost(src_eidl, dst_eidl, dim, True))
    else:
        if dim == 0:
            return 1
        else:
            return Size(src_eidl[dim - 1])
                * cost(src_eidl, dst_eidl, dim - 1, True)
```

An example output of this function is as below, for the case of the MIMO-IDFT scenario in Figure 2.3:

```
If(r_mimo_0 == r_idft_0, 0, Size(r_mimo_0)*Size(cw))
```

This is a Z3 If object which is evaluated in the Z3 solver: if the first element of the MIMO Decoder RDL is equal to the first element of the IDFT RDL, then the cost of the reorder buffer is zero; otherwise, it is the size of the MIMO RDL element 0 multiplied by the size of the (cw) dimension, which is element 0 of the MIMO IDL. In more complex scenarios, the size of the output grows. For the MIMO-IDFT-CD scenario, one constraint is generated for each of the two connections, and the output is as follows:

```
If(r_mimo_0 == r_idft_0,
   If(r_mimo_1 == r_idft_1,
      0,
      Size(r_mimo_1)*Size(cw)),
   Size(r_mimo_0)*Size(r_mimo_1)*Size(cw)),
If(r_idft_0 == r_cd_0,
   If(r_idft_1 == sym,
      0,
      Size(r_idft_1)*Size(sc)),
   Size(r_idft_0)*Size(r_idft_1)*Size(sc))
```

6.4.3 Solver output

The use of the solver to determine RDLs for simple scenarios will now be demonstrated. For the example in Figure 2.3, the solver produces the output in Listing 6.4, and the size of the required buffers (which may be used to guide system implementation decisions) and the execution time of the solver are summarised in Table 6.3.

Listing 6.4: Output from solver when applied to the MIMO/IDFT system.

```
Found a satisfying interpretation with total cost: 48
Optimal buffer cost is 48 with total data rate 48
Repetition dimension lists:
mimo: [sc]
idft: [cw]
Cost of buffer between mimo:w and idft:din is 48
```

Table 6.3: RDL results for MIMO/IDFT system.

Total buffer cost	Solver iterations	Execution time (s)
48	1	0.231

As expected, the solver has determined that the first block has an RDL of (sc) and the second block has an RDL of (cw) . This was determined in an execution time of 0.231 seconds, indicating that the execution time of the solver is short enough that it could be performed automatically in an interactive tool.

For the example in Figure 6.2, the solver finds a poor solution first, then finds a better solution which it cannot improve upon, as shown in Listing 6.5 and Table 6.4. The execution time remains low, at 0.238 seconds.

Listing 6.5: Output from solver when applied to the MIMO/IDFT/Channel Decoder system.

```
Found a satisfying interpretation with total cost: 1344
Found a satisfying interpretation with total cost: 672
Optimal buffer cost is 672 with total data rate 1344
Repetition dimension lists:
mimo: [sc, sym]
idft: [cw, sym]
cd : [cw]
Cost of buffer between mimo:w    and idft:din  is 672, data rate is 672
Cost of buffer between idft:dout and cd:din    is 0, data rate is 672
```

Note that the RDL on the MIMO decoder is different to that in Figure 6.3: both orderings produce the same buffer requirements.

Table 6.4: RDL results for MIMO/IDFT/Channel Decoder system.

Total buffer cost	Solver iterations	Execution time (s)
672	2	0.238

6.5 Inferring efficient data ordering to assist in component implementation

The discussion in this chapter so far has focused on the optimisation of memory requirements and latency in multidimensional data processing systems. In this section, another issue encountered in LTE system design is addressed, which is that the implementation of custom components is governed by the dimension ordering on upstream and downstream components. This problem was introduced in the context of the Uplink Resource Demapper in Chapter 2, and a diagram was provided in Figure 2.5.

To derive the most appropriate ordering of dimensions for the output interfaces of the Resource Demapper, a variant of the techniques described previously can be applied. Initially, the dimension list of the REF output on the Resource Demapper is set to be empty: in other words, it is regarded as outputting a single dimensionless token on each firing. Then SDF is used to determine the set of dimensions in the repetition list, and

then the Z3 solver is run to determine the most efficient ordering of those dimensions. The input to the solver is as follows:

```
comps = {
  'rd' : {'idl': {'ref': [], 'data': []},
         'rds': [sc,ant,sym]},
  'ce' : {'idl': {'din': [sc], 'dout': [cw,sc]},
         'rds': [sym,ant]},
  'mimo': {'idl': {'h': [cw,ant], 'y': [ant], 'w': [cw]},
         'rds': [sym,sc]},
  'idft': {'idl': {'din': [sc], 'dout': [sc]},
         'rds': [sym,cw]},
  'cd' : {'idl': {'din': [sym,sc]},
         'rds': [cw]}}

connections = [
  [['rd','ref'], ['ce','din']],
  [['ce','dout'], ['mimo','h']],
  [['rd','data'], ['mimo','y']],
  [['mimo','w'], ['idft','din']],
  [['idft','dout'], ['cd','din']]]
```

This produces the results in Table 6.5, and the proposed RDLs on some of the blocks are shown in Figure 6.4.

Table 6.5: RDL results for RD/CE/MIMO/IDFT/CD system.

Total buffer cost	Solver iterations	Execution time (s)
912	4	0.268

With no dimensions on the output interfaces of the Resource Demapper, the tool has proposed an RDL of (sym, sc, ant). However, this solution requires a buffer on the REF output of the Resource Demapper, which is not necessary if the IDLs on the Resource Demapper can be modified instead of the RDL. To allow this, an *interface dimension set* (IDS) can be described for each interface on each component, into which the user may optionally migrate some of the RDS dimensions that are generated by symbolic SDF, and this allows the solver to determine efficient IDLs instead of RDLs where an IDS is provided instead of an RDS. The input to the tool has the same inputs as before, except that the contents of the RDS for the Resource Demapper have been migrated into an IDS for each of its interfaces, as follows:

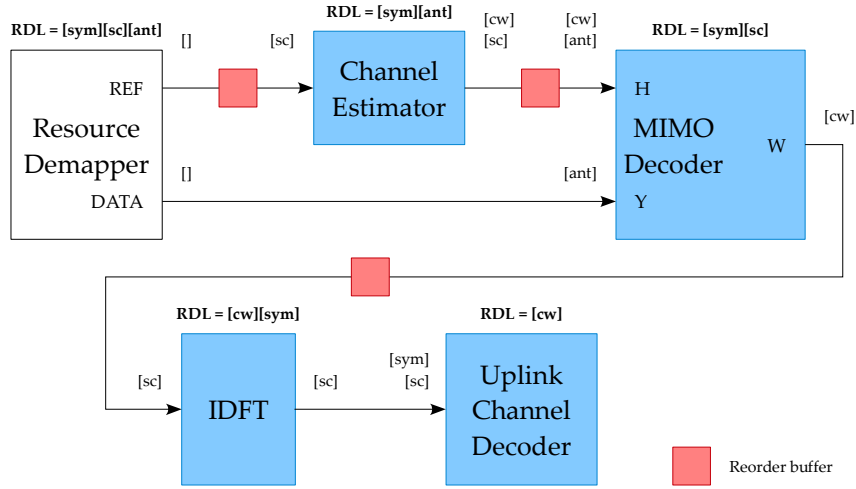


Figure 6.4: Efficient RDL calculated for Resource Demapper.

```
'rd': {'ids': {'ref': [sym, ant, sc], 'data': [sym, ant, sc]}, 'rds': []}
```

The proposed RDLs and IDLs in the modified system are shown in Figure 6.5 and summarised in Table 6.6. While the tool requires a greater number of iterations to arrive at its final result, this does not affect the execution time significantly.

Table 6.6: RDL/IDL results for RD/CE/MIMO/IDFT/CD system.

Total buffer cost	Solver iterations	Execution time (s)
864	8	0.290

This version has proposed an improved solution with a buffer cost of 864 rather than 912, by allowing the IDL to be modified instead of the RDL.

6.6 Eliminating redundant calculations

In the previous example, it was determined that the (sym) dimension is required on the REF output of the Resource Demapper. An intuitive explanation is that the MIMO Decoder requires one estimate per data symbol resource element, and thus the Channel Estimator must produce a channel estimate for each data symbol resource element, and thus the Resource Demapper must produce a reference symbol resource element for each

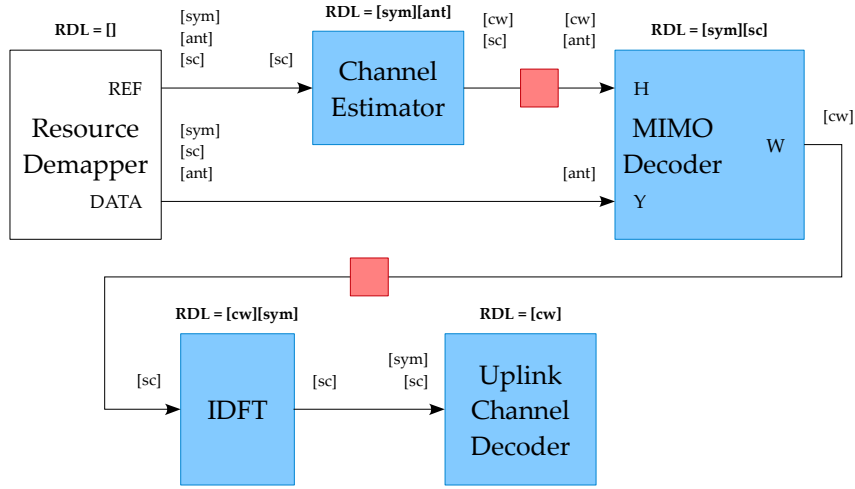


Figure 6.5: Calculation of IDLs from IDSs instead of RDL from RDS for the Resource Demapper leads to lower buffering requirements.

data symbol resource element.

However, as shown in Figure 2.2, there is only one reference symbol in each slot. One way to address this issue is to replicate the reference symbol data for each slot in the Resource Demapper. However, this means that a channel estimate must be calculated repeatedly for the same reference symbol, which imposes unnecessarily high performance requirements on the Channel Estimator.

To avoid this recalculation, a data replicator block may be introduced between the Channel Estimator and the MIMO Decoder. This block consumes a single reference symbol data element and produces a copy of that element for each data symbol in the current slot. Adding this block to the SMT solver's input system, introducing a (slot) dimension and running the solver again results in the solution shown in Figure 6.6 and summarised in Table 6.7. Notably, the execution time of the solver has increased significantly in this scenario, and further research is required in order to assess the reasons for this and its implications. It should also be noted, however, that since the number of solver iterations has not increased significantly, the increase in execution time cannot be blamed solely on poor reuse of previous SMT lemmas in the optimisation process.

In this solution, buffers are required on each side of the replicator, but this is not how the system would be designed in a manual implementation. Instead, it would be better to have a single buffer between the Channel Estimator and the MIMO Decoder and for the

Table 6.7: RDL results for RD/CE/MIMO/IDFT/CD system with symbol replicator.

Total buffer cost	Solver iterations	Execution time (s)
976	13	0.957

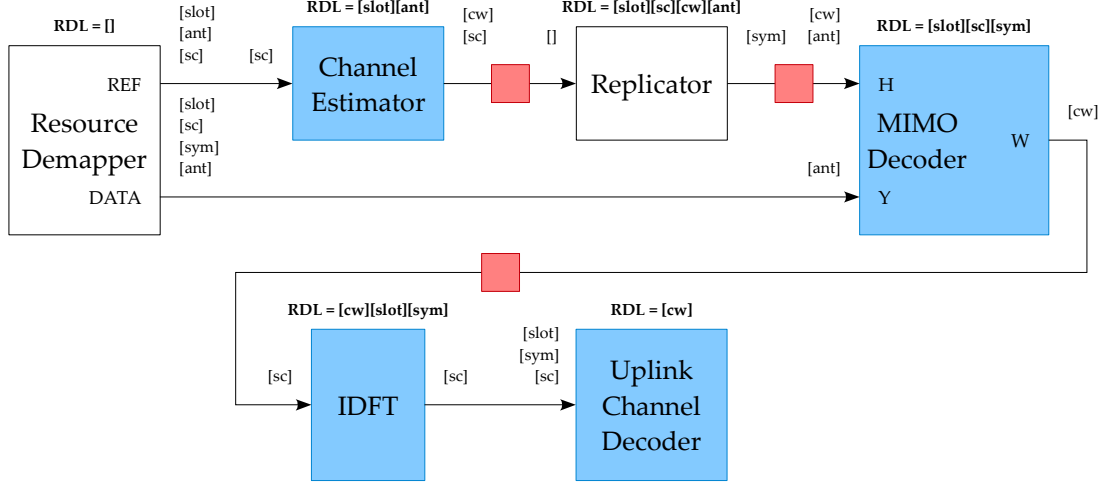


Figure 6.6: RD/CE/MIMO/IDFT/CD system with symbol replicator.

(sym) dimension to be replicated within that buffer.

If this combined reorder-and-replicate block were implemented manually, the required IDLs could be determined in a similar manner to the IDLs for the Resource Demapper as described in Section 6.5: the dimensions in the RDL for the replicator could be moved to the IDLs. However, in this scenario, the reorder-and-replicate block is moved outside of the domain of applicability of the SMT solver and future changes to the system may require that the block is reimplemented, since the dimension orderings in the system could change.

Since the reorder-and-replicate block is fairly simple, it would be desirable for it to be generated and instantiated automatically, allowing its memory cost to be reflected in the results of the optimisation process. This has not been implemented, but it could be done by removing the (sym) dimension from the RDS of the Channel Estimator, as shown in Figure 6.7. The SMT solver will run and produce a correct calculation of the buffering cost, as long as the buffer cost function is updated such that it only considers dimensions present in both lists. Once the most efficient dimension orderings have been determined,

the block could be generated automatically using a similar method to that described in Section 6.2.2, with similar code produced except that the data copying statements would be generated as shown in Listing 6.6, without buffering the (sym) dimension.

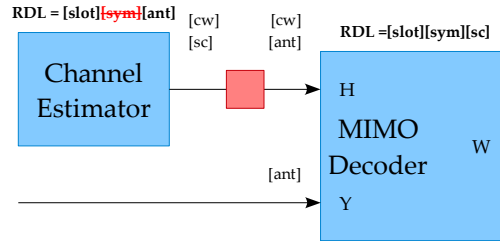


Figure 6.7: The (sym) dimension removed from the Channel Estimator.

Listing 6.6: Modified data copying statements for input to Vivado HLS.

```
buf[slot][ant][cw][sc] = in[slot][ant][cw][sc];
...
out[slot][sc][sym][cw][ant] = buf[slot][ant][cw][sc];
```

6.6.1 Generalisation to arbitrary generation and reduction functions

The flow as described so far assumes that a dimension present in an output list but not an input list should be generated through replication of elements, but this is not always the best approach: a variety of generative (anamorphic) functions may be used. A dual problem occurs when a dimension appears on an input but not an output, since in this case a variety of reductive (catamorphic) functions can be applied. This problem is highly relevant to the uplink receive system, since improved performance can be achieved by replacing the replicator function with a block that generates an interpolated channel estimate for each symbol from the reference symbols in each slot; in other words, the replicator with a type signature of $() \rightarrow (sym)$ becomes an interpolator of type $(slot) \rightarrow (slot, sym)$. As before, two integration options are initially apparent:

1. instantiate an interpolator block manually, but with the problem that some dimensions are reordered unnecessarily on the input and output connections of this block.
2. determine the required dimensions on the interfaces of a combined reorder-and-interpolate block and implement this manually, with no unnecessary dimension

reordering.

If we no longer assume that elements in new dimensions must be generated through replication, another option becomes possible if the generation process is specified:

3. indicate to the tool that the 'sym' dimension is generated from the 'slot' dimension, provide an interpolation function which does this, and have the tool generate the combined block automatically.

An example of a mechanism for specifying this function is shown in Figure 6.8, and the code that would be generated as input to an HLS tool is shown in Listing 6.7.

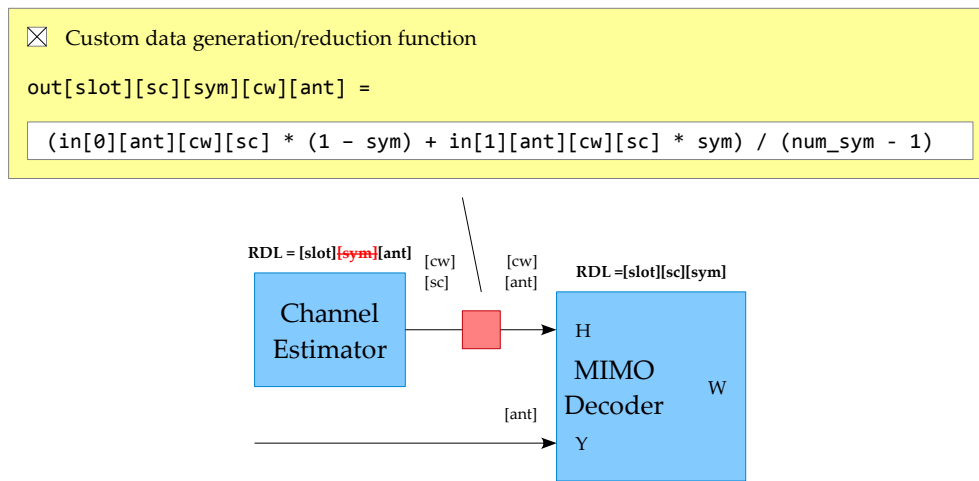


Figure 6.8: GUI mock-up of a mechanism for specifying arbitrary data generation and reduction functions.

Listing 6.7: C code for custom replicate-and-reorder buffer to be input to Vivado HLS.

```
for (slot = 0; slot < num_slot; slot++)
  for (sc = 0; sc < num_sc; sc++)
    for (sym = 0; sym < num_sym; sym++)
      for (cw = 0; cw < num_cw; cw++)
        for (ant = 0; ant < num_ant; ant++)
        {
          out[slot][sc][sym][cw][ant] = (in[0][ant][cw][sc] * (1 - sym) +
            in[1][ant][cw][sc] * sym) / (num_sym - 1)
        }
```

6.7 Implementation considerations

There are some potential limitations with the approach described in this chapter that must be addressed. Earlier, it was mentioned that the IDFT operates on subcarriers, while the metadata for the core would instead refer to these as “elements” or similar. Since we need the “elements” dimension to be interpreted as “subcarriers” in the LTE context, a mechanism is required for mapping between dimension names. This could be done when components are connected: if both components have a dimension that isn’t present on the other component, a dialog box could offer a “patch panel” consisting of a list of dimensions on both components, as demonstrated in Figure 6.9, with the user instructed to define the appropriate mappings.

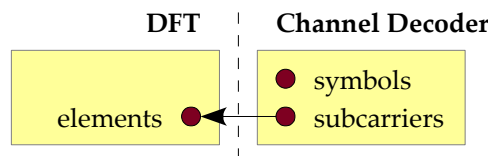


Figure 6.9: GUI mock-up showing the association of the “subcarriers” dimension in the Channel Decoder with the “elements” dimension in the IDFT in a patch panel, with the direction of the arrow indicating that the associated dimensions should be named “subcarriers”.

Another issue is that the approach described here assume static system-wide dimension sizes, while in reality, these are often fully dynamic, or are sized differently in different parts of the system. However, since signal processing systems are often designed according to their worst-case behaviour, the process may be applied instead to maximum dimension sizes where these are recorded in metadata. To deal with changing dimension sizes, they could be renamed in different parts of the system.

6.8 Discussion

The optimisation process discussed here is implemented separately to the design flow that is required to implement the design that the solver proposes. One advantage of this arrangement is that the solver will not interfere with status quo design flows in tools like System Generator, and can simply be consulted when necessary to provide additional

input into the manual design process. However, one drawback is that additional manual effort is required to keep the solver's view of the system synchronised with that of the baseline design tool. Thus, the flow could be improved if the solver were integrated into a design environment such as Vivado IP Integrator, which would allow greater ease-of-use and would potentially improve system implementation time. Some care would be required, however, to ensure that the tool can be overridden if necessary.

6.9 Conclusion

This chapter has set out to address the issues encountered in constructing multidimensional signal processing systems. An approach has been presented which determines the need for reorder buffers using the metadata descriptions of the interface data types in the system, and a number of ways to generate these buffers automatically have been discussed, including chaining of SID cores and generation using a high-level synthesis tool.

Furthermore, an approach has been presented for determining efficient buffering scenarios from a variety of possible options using an SMT solver. These techniques were applied to the Xilinx LTE Uplink system, and were also used to determine efficient interface data orderings on the Resource Demapper. While this automated approach will not always allow a more efficient system to be built than could be created by hand, it allows efficient buffering scenarios to be proposed within seconds of laying out a design, rather than the minutes or hours that could be required in a manual approach.

Finally, some limitations of this approach were presented, and solutions were proposed. Firstly, dimensions can sometimes be propagated unnecessarily, but this can be addressed by manually overriding the calculated dimension lists. Secondly, manual instantiation of blocks such as interpolators can introduce unnecessary data reorderings, and thus it is desirable to merge the interpolation and reordering functions; a proposed solution is to provide a custom data processing function to be performed in reorder buffers. Finally, an approach for mapping dimension names was proposed.

Chapter 7

A software model generation framework based on extended IP-XACT

Earlier chapters in this thesis discussed the benefits of generating the system-level concerns of software models and heterogeneous hardware systems from the same system description, and it was proposed that IP-XACT, together with extensions defined in Chapter 5, can be used to represent the required system information in metadata. In this chapter, it will be shown how this metadata may be used as an intermediate representation in an end-to-end toolflow that converts high-level dataflow descriptions of LTE systems to this IR, and it will be shown how this IR can be used to generate code for one of the intended target platforms, which is the software simulation model of the system that was originally written manually. Both the downlink transmit and uplink receive LTE system simulation models have been implemented in this framework, but since the implementation process in each case was similar, this chapter focuses only on the first of these.

The model generation process will be demonstrated by presenting each toolflow feature as it is required in the process of implementing successively larger portions of the original model. The structure of the original XMODEL system is broadly similar to the structure shown in Figure 2.1, and any significant deviations from this structure made in the

dataflow abstraction process will be discussed.

The dataflow properties of the components are represented in the CAL language, but while code generation back-ends for CAL already exist, the toolflow must remain independent of any particular high-level input language such as CAL. The aims of the work in this chapter are to demonstrate solutions to the engineering challenges of, firstly, building a dataflow code generation flow around an intermediate representation based on the industry standard IP-XACT XML schema; secondly, representing an industrially relevant application using a dataflow abstraction; and thirdly, integrating the flow with existing Xilinx tools and modelling environments.

7.1 Simple leaf-level components

The first components to be discussed are `modulate` and `scramble`, since they are some of the simplest components in the LTE downlink system.

7.1.1 Input language

In the original LTE downlink transmit model, the `modulate` component has three ports named `din`, `dout` and `ctrl`, and executes a function called `process`. It can be represented in a CAL definition using an action which consumes tokens on the `din` and `ctrl` ports and produces tokens on the `dout` port, via an output expression that calls the `process` function, as follows:

```
package xlte.dl_tx;

import all modulate_types;

actor modulate ()
  modulate_din din, modulate_ctrl ctrl ==> modulate_dout dout:

  action din: [a], ctrl: [c] ==> dout: [process(a, c)] end
end
```

The CAL description for the `scramble` component is correspondingly simple. These CAL descriptions are converted to the extended IP-XACT IR using a parser provided in the

OpenDF project which outputs an abstract syntax tree in the XDF schema, which is then transformed into the IR using XSLT. The following paragraphs describe specific aspects of this transformation process.

IP-XACT component descriptions must be identified by a vendor, library, name and version (known as a VLNV), and none of these identifiers may be omitted or left blank. Of these identifiers, CAL can be used to specify only a component (actor) name and package, so the CAL component name is written to the IP-XACT name field, and the CAL package is written to the IP-XACT library field. For the IP-XACT vendor, I use “xilinx.com” and for the IP-XACT version, I use the value “1.00.a” since this is a default version number used elsewhere in Xilinx.

```
component:
  vendor: xilinx.com
  library: xlte.dl_tx
  name: modulate
  version: 1.00.a
```

For each of the ports in the CAL description, an IP-XACT bus interface is generated: in this implementation there is a direct mapping from dataflow ports onto RTL-layer bus interfaces¹. The `modulate` component consumes tokens of type `modulate_din` and `modulate_ctrl` and produces tokens of type `modulate_dout`, and these types are associated with interfaces using the `dataTypeRef` element as described in Chapter 5. Since each `dataTypeRef` requires a VLNV, a library must be found for each type. One option is to define the library explicitly in the CAL description, for example: `import modulate_types.modulate_din`, which indicates that the `modulate_din` type is found in the `modulate_types` library, or, to avoid repetition, the `import all modulate_types` syntax may be used. In the latter case, a search is performed in all XML type libraries that are included using the `import all` syntax in order to determine the associated library. Since CAL generates an IR in XML form (XDF) and the type libraries are described in XML, this process can be expressed in XSLT as follows:

¹The multiplexing of dataflow channels over physical channels could be investigated in future work.

```

<xsl:variable name="type-candidates">
  <xsl:for-each select="Import[@kind = 'package']">
    <xsl:copy-of select="document(concat(QID/ID/@name, '/types.xml'))
                      //xilinx-dsp:dataTypeDef
                      [spirit:name = $tname]"/>
  </xsl:for-each>
</xsl:variable>

<xsl:if test="$type-candidates/xilinx-dsp:dataTypeDef">
  <xsl:value-of select="$type-candidates/xilinx-dsp:dataTypeDef[1]
                      /spirit:library"/>
</xsl:if>

```

The first part stores all of the data type definitions (`xilinx-dsp:dataTypeDef`) that have the requested name (`[spirit:name = $tname]`) from the libraries imported using the `import` all syntax (`Import[@kind = 'package']`), and the second part assigns the result to be the library of the first of these if the resulting list is non-empty. If the type cannot be found in any included libraries, then the XSLT processor exits with an error (although this is not shown in the example above).

7.1.2 XMODEL code generation

Once an IR has been produced for the `modulate` component, the XMODEL component code that wraps the process function may be generated automatically. Code can be generated from an XML IR using a variety of tools, including XSLT and the Perl Template Toolkit 2 (TT2). XSLT is standardised as a Recommendation of the World Wide Web Consortium (W3C), while TT2 is a template language that is more flexible than XSLT since it allows the execution of arbitrary Perl code where necessary. While I use XSLT for most XML-to-XML transformations, the additional flexibility of the TT2 approach is often useful in the generation of executable code, for example when XPath expressions must be manipulated directly (as will be discussed later) and thus I use TT2 templates to generate XMODEL code.

XMODEL components are defined in the form of C++ classes. For each component in the system, whether it forms a leaf-level component or a hierarchical component, a C++ header (`.h`) and implementation (`.cc`) file must be generated. A TT2 template consists of a sequence of directives such as the following, which generates a test node declaration

(introduced in Chapter 2) for each bus interface in the IP-XACT file:

```
[% FOREACH bi IN component.findnodes("spirit:busInterfaces
                                     /spirit:busInterface") %]
    xtestnode& m_tn_ [% bi.findvalue("spirit:name") %];
[% END %]
```

As described in Chapter 2, XMODEL input interfaces provide a push function and output interfaces provide a pop function and sometimes a peek and/or empty function, though the contents of these functions are not standardised and are implemented manually: the main processing function (such as the process function in the `modulate` component) may be called by either the push or the pop function. The model of computation used by XMODEL components, involving a combination of data-driven and demand-driven communication involving token pushes and pulls, corresponds approximately to the Component Interaction (CI) domain in Ptolemy [119].

Component Interaction employs a less rigid model of computation than the dataflow domains such as SDF and DDF, and as a consequence, the code generation opportunities are more limited. By standardising the implementations of the push, pop, peek and empty functions, XMODEL components may be generated automatically. An XMODEL FIFO and test node are generated for each interface², and component processing is governed by generated *action functions* which test whether an action has become fireable, and execute a processing function if so.

Each action function returns a boolean value: TRUE to indicate that the action was fireable and has thus fired, or FALSE to indicate that the action was not fireable. The processes involved in each action function are as follows:

1. Determine the number of tokens required on each input port, which in the case of the `modulate` and `scramble` blocks is a static value specified in the IP-XACT code;
2. Declare variables for all of the action's input and output tokens;

²While a policy could be adopted in which FIFOs are provided only on the input or output interfaces, providing FIFOs on both inputs and outputs avoids the need for a component to maintain references to the source or destination components to which it is connected via its interfaces. Instead, the references that define the topology of a subsystem are held solely by its parent hierarchical component.

3. Check FIFOs to determine whether the required number of tokens are present: if not, return FALSE;
4. For ports requiring more than one token, resize the vector in order to hold the right number of tokens;
5. Pop tokens from input FIFOs into pre-declared variables;
6. Execute procedural action statements;
7. Determine values of output expressions and write these to output tokens;
8. Push generated output tokens to output port FIFOs;
9. Return TRUE.

A number of actions may be associated with each actor, and thus a function is required which calls each of the action functions in turn until no action is fireable. This function is called `fire_all`. Calls to the `fire_all` function are added to the push function, so that every time a new data token arrives, the various firing rules are checked to determine whether an action may fire. Another call to `fire_all` is also added to the empty function, which provides a means to invoke actions that require no input tokens, as are found in data source components, for example.

Actions in CAL are optionally named, and if no name is provided for an action then a unique name is generated for its associated action function: `action_0`, `action_1`, and so on; the `modulate` actor has a single action which is given the name `action_0`. The output expression in this action is an invocation of the `process` function. Since we wish to reuse an existing C++ definition of this function, it is not defined in the CAL description, and instead of converting a CAL function into C++ code, a C++ function declaration is generated in the actor's class definition by determining which tokens are processed by the function:

```
const modulate_dout process(const modulate_din a, const modulate_ctrl c);
```

A function definition corresponding to this declaration must then be provided in a separate C++ file, which will be called the *user-defined function file*, and this sits alongside the

CAL definition of the actor, using the same filename but with a `.cc` extension. Whenever such a file is present, its contents are copied automatically into the generated C++ code.

While the `modulate` actor has a single action which takes one token from each input port and writes one token to the output port, other components have actions which consume and produce multiple tokens, and this is indicated using the `repeat` keyword in CAL. For example, if the `modulate` action consumed three tokens from `din` and wrote two tokens to `dout`, the action definition would be as follows:

```
action din:  [a] repeat 3, ctrl: [c]
  ==> dout: [modulate(a, c)] repeat 2
end
```

In the generated software code, declarations of functions that read or write multiple tokens on an interface are provided with a vector parameter or return value, as follows:

```
const std::vector<modulate_dout>
  process(const std::vector<modulate_din> a, const modulate_ctrl c);
```

While the token counts on each interface are static in many cases, they are currently determined dynamically in the software code by requesting the size of the `std::vector` objects.

7.2 Data type input and code generation

This section describes how data types as represented in the IR described in Chapter 5 are used to generate C++ code in the XMODEL framework. These flows are shown in Figure 7.1 and explained below.

7.2.1 Input language

In order to parse high-level descriptions of data types, I have implemented parsers from two existing high-level languages using the ANTLR parser generator [120]. The first uses a freely-available grammar for the ASN.1 language³ to generate a parser for ASN.1 data

³Available at <http://www.antlr3.org/grammar/list.html>

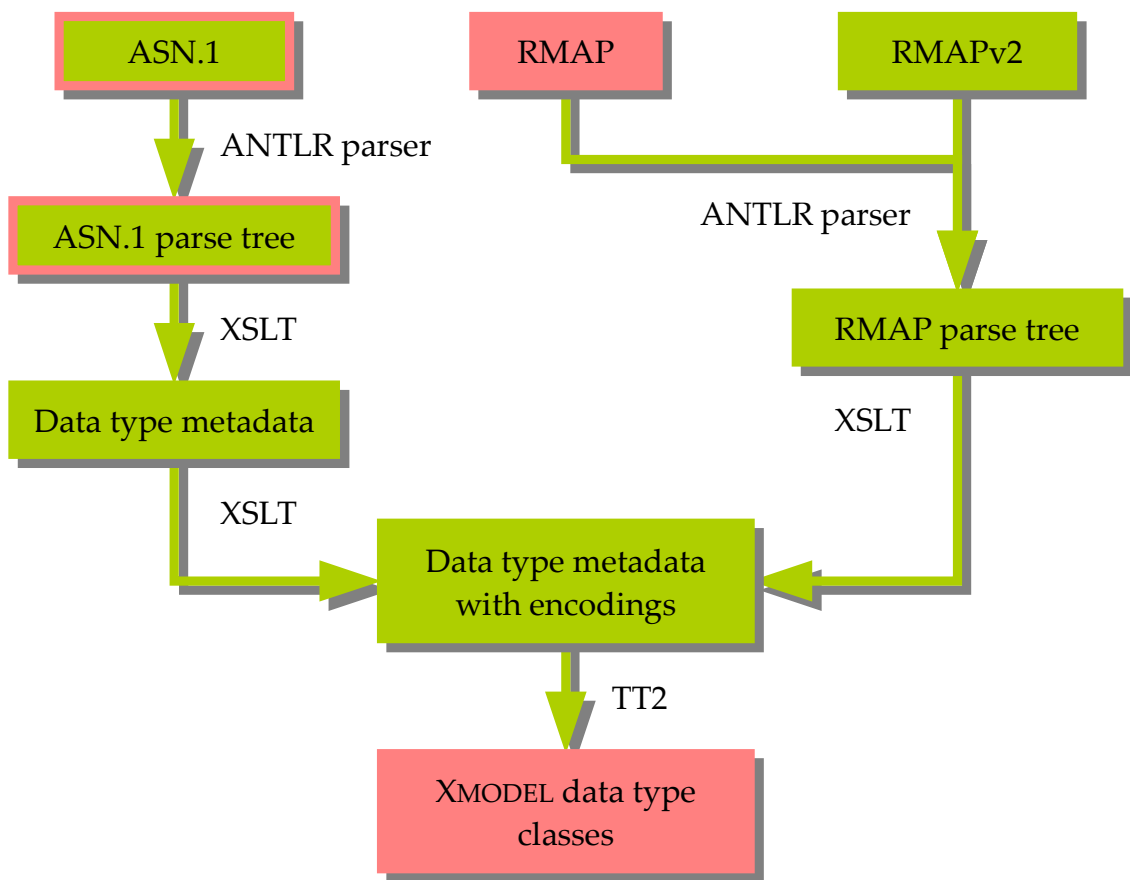


Figure 7.1: Data type metadata and code generation flow (the significance of the colours is as shown in Figure 4.3).

type descriptions, and I have extended the generated parser such that it outputs parse trees in XML form. Further XSLT processing then converts this to XML code without data type encodings, with a further XSLT processing stage generating encodings corresponding approximately to the ASN.1 Packed Encoding Rules standard [110]. Since ASN.1 cannot be used to describe data type encodings, this flow may be used to describe data types in situations where a common encoding of the types must be shared between two or more components, but in which the specific encoding is not important.

When integrating IP cores, the specific encoding of the data types is important, so a high-level language must be used which is able to describe encodings concisely. An appropriate language may be derived through suitable extensions to an existing language used within Xilinx called RMAP, which is used to describe register maps. An example of an RMAP description from the pre-existing downlink transmit system is as follows:

```

REGMAP dl_tx_sch_qam_ctrl    # Downlink Transmit SCRAMBLE/QAM/MIMO Control
  REGGROUP qam               # Modulation
    REG mod1                 # Modulation 1
      FIELD qam      1:0     # QAM Modulation Type(only valid in SCH QAM)
      FIELD n_cbs 15:8      # Number of Code Blocks in Transport Block
  REGGROUP scrambler         # SCRAMBLER
    REG scrambler0           # PDSCH uses flexible cinit
      FIELD c_init0 31:0     # C init0 value.
    REG scrambler1
      FIELD c_init1 31:0     # C init1 value.
  REGGROUP mimo              # MIMO
    REG mimo_conf            # MIMO configuration(check against core)
      FIELD sm_or_td 0:0     # spatial vs transmit diversity
      FIELD number_codewords 9:8 # required for matrix selection
      FIELD codebook_index 19:16 # required for matrix selection

```

In the existing LTE systems, a parser written in Perl is used to convert these descriptions into C and HDL for use as a hardware abstraction layer between layers 1 and 2 of each system. In order to conform to the metadata-based flow described in this thesis, these RMAP descriptions must be converted into XML to be added to IP-XACT descriptions, and these in turn must be converted to C++ for use with the XMODEL LTE systems. For this purpose, I created an ANTLR grammar for the RMAP language which is used to generate an RMAP parser, and as in the ASN.1 case, this parser outputs an XML parse tree which is converted to the standard XML metadata format described in Chapter 5. How-

ever, since RMAP data type encodings are user-specified, they need not be automatically generated in this process.

In the RMAP language, registers do not have types associated with them and the type hierarchy is not supported, so an extended version of this language was created and will be referred to as RMAPv2. The RMAPv2 language is intended to be backwards-compatible with the existing RMAP language, and therefore the newly-created RMAP grammar was modified in order to admit descriptions both in RMAP and RMAPv2 syntax. Modifications made in RMAPv2 include the following:

- description of an abstract TYPE rather than the REGMAP which is oriented towards memory-mapped interfaces;
- description of packet structure using STRUCTURE, UNION and ARRAY keywords;
- identification of the type of leaf-level elements, such as COMPLEX or INT with a specified range.
- optional field names;
- description of the encoding of particular values in a field such as NANT in the PUCCH; and
- type references, allowing aggregation of control packets.

An example of an RMAPv2 description which describes the data packet used in the wrapped, single-channel XFFT core (an array consisting of complex values with 14-bit fractional widths) is shown below:

```
TYPE xfft_v8_0_wrapper_data_packet
  ARRAY elements
    FIELD 31:0 COMPLEX(14)
```

7.2.2 XMODEL code generation

XMODEL supports basic types such as `xuint32` and `xbit`, multidimensional arrays such as `xmatrix_uint32` and `xmatrix_bit` and control packets which are subclasses of an `xcontrol_packet` class. `xmatrix` is a class that was created in the original XMODEL

framework to represent multi-dimensional arrays, and is used in preference to nested C++ STL vectors because it can guarantee that the memory used is contiguous and that all sub-vectors are the same size.

For simple types in the IR (integers and reals, for example), a C++ typedef to a pre-defined XMODEL type is generated, and for arrays of simple types, a typedef to the appropriate XMODEL xmatrix type is generated. Integer types with a restricted set of enumerated values generate a C++ enumerated type. For structures in the IR, a class definition is generated. The following example is an LLR control packet which has been automatically generated for the uplink model, containing four fields that are accessed through get and set methods. The use of these methods to interact with fields represented as private class members allows validation of the values written to those fields in the corresponding .cc file.

```
namespace xmodel
{
  class llr_ctrl_packet: public xcontrol_packet
  {
  public:
    llr_ctrl_packet() {}

    /* Mutator methods. */
    void set_modulation(const xuint32 data);
    void set_inv_sigma_sq(const xuint32 data);
    void set_init_x_1(const xuint32 data);
    void set_init_x_2(const xuint32 data);

    /* Accessor methods. */
    const xuint32 get_modulation() const;
    ...

  private:

    /* Struct fields. */
    xuint32 modulation;
    xuint32 inv_sigma_sq;
    xuint32 init_x_1;
    xuint32 init_x_2;
  };
};
```

The v_append_packet function from the .cc file, which serializes the type to an array of

integers, is shown below. A `v_parse_packet` function is also present in the file, and this performs the inverse operation.

```
void llr_ctrl_packet::v_append_packet(xuint32_packet& p) const
{
    size_t s;

    s = p.size();
    p.resize(s + 1);
    p[s] |= (get_modulation() & (1 << 2) - 1);
    p[s] |= (get_inv_sigma_sq() & (1 << 16) - 1) << 16;

    s = p.size();
    p.resize(s + 1);
    p[s] |= (get_init_x_1());

    s = p.size();
    p.resize(s + 1);
    p[s] |= (get_init_x_2());
}
```

Additional information in the metadata can be used to further refine the generated code: for example, fields that are marked as being optional could generate a presence flag with an accessor method and a method to clear the field, although this flow does not currently use dependencies to determine the presence of optional fields since this is not required in the LTE systems.

7.3 Integrating bit-accurate core simulation models

In contrast to the modulate and scramble blocks, the Channel Encoder and MIMO encoder already have a function defined in a bit-accurate simulation model, and instead of writing a custom function in C++, the existing function provided by the bit-accurate model must be integrated.

Xilinx software simulation models are provided in the form of a static object (Linux) or dynamic linked library (Windows), with a C header file defining an API with which to use the model. Integration is achieved by adding wrappers to the models so that they expose homogeneous XMODEL interfaces, which was done in the original LTE system

simulation models, but as with the other XMODEL LTE components, much of the code in these wrappers may be generated by extending the IP-XACT metadata associated with the LogiCORE with references to the entry points of those wrappers.

The IP-XACT files for LogiCOREs are stored in subdirectories of the IP Catalog path, with a directory name that is derived from the core's name and version. For example, the MIMO Encoder v2.0 is stored in `lte_3gpp_mimo_encoder_v2_0/component.xml`. To describe the dataflow properties of this core, a CAL description is created with the name `lte_3gpp_mimo_encoder_v2_0.cal` and this is used to generate another IP-XACT file which is then merged with the original IP-XACT file from the IP Catalog. This flow is shown in Figure 7.2.

There are two aspects of component specification that require a different procedure when core simulation models are being wrapped: the dataflow actions, and the parameters.

7.3.1 Actions

Most LogiCORE simulation models have a single `bitacc_simulate` function which usually processes a single input data packet and control packet into an output data packet. These functions are wrapped with a dataflow action that is defined in CAL in the same way as for custom components. For example, the MIMO Encoder has the following action:

```
action din: [a], ctrl: [c] ==> dout: [process(a, c)] end
```

The `process` function is defined in C++ in the user-defined function file as before, but it is implemented such that it wraps the simulation model by converting the input data stored as instances of XMODEL classes into the C structures that are used by the LogiCORE model, calling the `bitacc_simulate` function, and converting the output data back into the XMODEL format.

The Channel Encoder simulation model has three `bitacc_simulate` functions: one for the PDSCH, one for the PDCCH and PBCH, and one for the PCFICH and PHICH. This component can be modelled as a dataflow actor with three actions, each of which con-

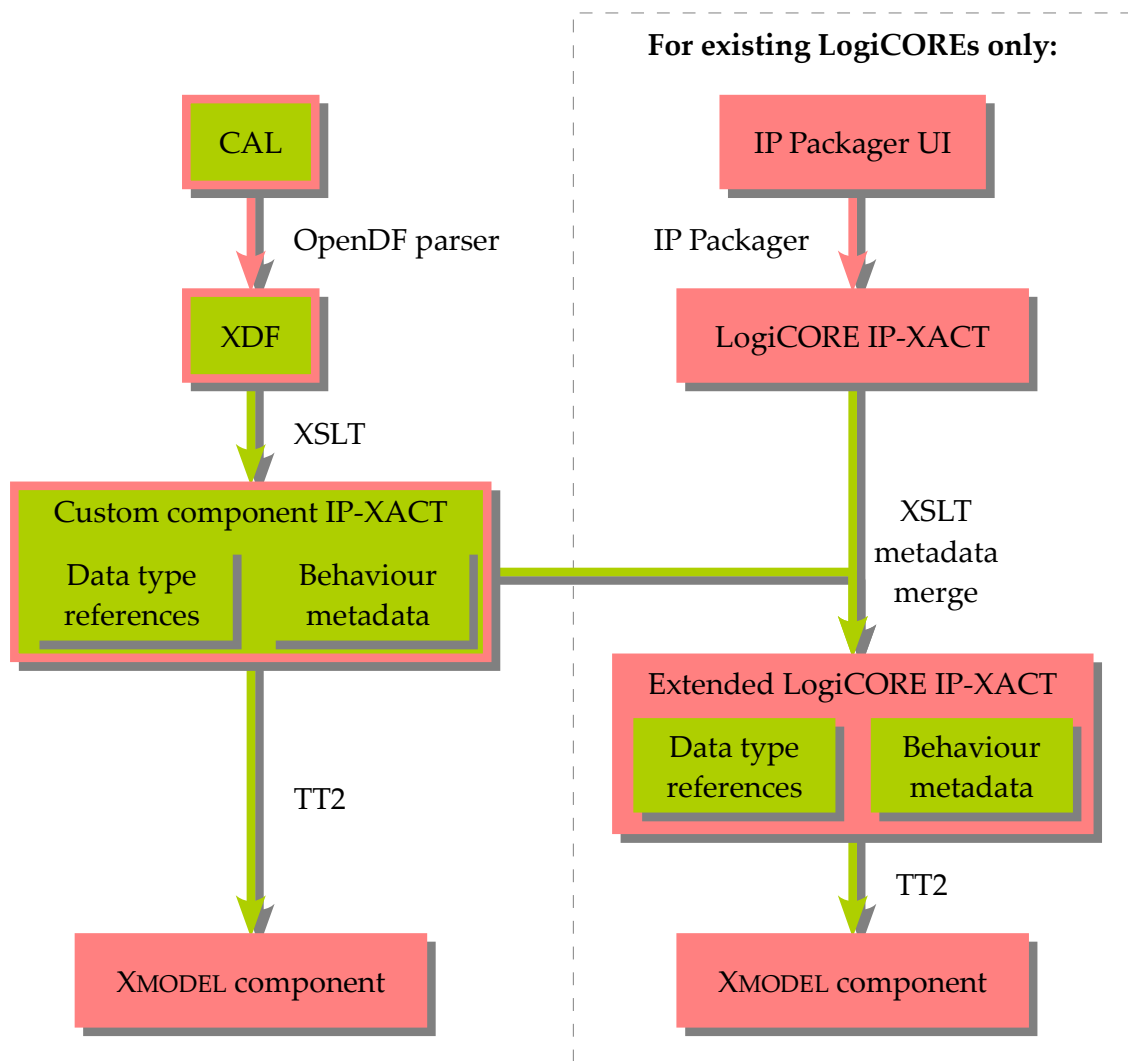


Figure 7.2: Component metadata and code generation flow (the significance of the colours is as shown in Figure 4.3).

sumes a control packet and a data packet, except for the PCFICH/PHICH action which only consumes a control packet.

7.3.2 Parameters

When a LogiCORE simulation model is wrapped using the approach described in this section, parameters in the pre-existing IP-XACT for the hardware core are merged into the IP-XACT file. However, the software models typically use different sets of parameter names. In many cases, these parameters duplicate functionality in the hardware core: for example, the MIMO Encoder core has a `Number_of_Output_Antennas` parameter while the MIMO Encoder software model has a `C_NUM_ANT` parameter with the same meaning. In other cases, these parameters add new functionality: for example, the MIMO Encoder software model has a parameter named `C_PC_SCALING` which is not applicable in the hardware core.

To ensure that a complete but minimal set of parameters is included in the IP-XACT IR, the software parameters that are not duplicated by a hardware parameter are added to the CAL description: `C_PC_SCALING` would be included, but not `C_NUM_ANT`. The LogiCORE parameters and the software model parameters defined in CAL are merged into a single IR.

For each of the parameters in the IR, a `config_set_` function declaration is generated, and definitions of these functions are provided in the C++ user-defined function file. In the case of the MIMO Encoder parameters, a simple assignment is required:

```
void lte_3gpp_mimo_encoder_v2_0::config_set_C_PC_SCALING
    (const xuint32 C_PC_SCALING)
{ generics.C_PC_SCALING = C_PC_SCALING; }

void lte_3gpp_mimo_encoder_v2_0::config_set_Number_of_Output_Antennas
    (const xuint32 Number_of_Output_Antennas)
{ generics.C_NUM_ANT = Number_of_Output_Antennas; }
```

In other cases, more complicated relationships between hardware parameters and software model parameters must be accounted for, such as conversions between string and integer types:

```

void lte_3gpp_mimo_encoder_v2_0::config_set_Diversity_Multiplexing_Options
    (const std::string Diversity_Multiplexing_Options)
{
    if (Diversity_Multiplexing_Options == "Transmit_Diversity_Only")
    { m_generics.C_HAS_SPATIAL_MUX = 0; }
    else if (Diversity_Multiplexing_Options
        == "Transmit_Diversity_Spatial_Multiplexing")
    { m_generics.C_HAS_SPATIAL_MUX = 1; }
}

```

The data types of the parameters are specified in IP-XACT, allowing the correct method declaration to be generated for each parameter.

7.3.3 Fully automatic integration of software models

It would be desirable for the simulation model wrapping process to be performed automatically, but this is currently impossible: while the APIs of each of the core models look quite similar, minor differences prevent their automatic integration into systems. One difference is found in the conventions for representing hierarchy in data arrays and control packets. For example, the MIMO encoder simulation model output structure has eight fields for the real and imaginary components of each of four antennas in the Υ interface, named as $y0i$, $y0q$, $y1i$ and so on. The MIMO decoder, however, has fields named Y_I and Y_Q , with data from multiple antennas stored sequentially in these arrays. Another format is found in the XFFT, which has fields named xk_re and xk_im for an interface named `DATA`, and unlike the HDL core, no multi-channel behaviour is accounted for. The xk name in this case cannot be derived from metadata.

Another problem is the inconsistency between hardware and software parameters, which requires parameter-setting methods to be defined manually as described in Section 7.3.2.

To solve these problems, a unified C API must be created that covers both the data types and dynamic behaviour of the software models, and a proposal for such an API is detailed below. Firstly, all data arrays are represented as flat memory regions with initialisation functions and functions (or macros) that calculate address offsets given the indices in each dimension. This approach is suitable both for static and dynamic dimension sizes, although the arrays will need to be recreated where necessary when dimension sizes are

dynamic. Arrays of complex values are declared as arrays of structures containing a real value and an imaginary value, rather than as two separate arrays.

Furthermore, there should be a correspondence between the action output expressions in metadata and the simulation functions, which is currently impossible because a wrapper function is required.

Applying these proposals to the XFFT core results in an API such as the following:

```
typedef struct {
    int i;
    int q;
} complex_t;

typedef complex_t *xfft_v8_0_xk;
typedef complex_t *xfft_v8_0_xn;

xfft_v8_0_xk xfft_v8_0_xk_create(int n_channels, int n_elements);
xfft_v8_0_xn xfft_v8_0_xn_create(int n_channels, int n_elements);
void          xfft_v8_0_xk_destroy(xfft_v8_0_xk);
void          xfft_v8_0_xn_destroy(xfft_v8_0_xn);

void          xfft_v8_0_xk_write(xfft_v8_0_xk xk,
                                int channels_el,
                                int elements_el,
                                complex_t value);
complex_t     xfft_v8_0_xk_read(xfft_v8_0_xk xn,
                                int channels_el,
                                int elements_el);

void          xfft_v8_0_bitacc_simulate(const xfft_v8_0_xk xk,
                                         xfft_v8_0_xn xn)
```

If these proposals (or similar ones) were implemented across all cores, simulation models could be wrapped automatically, or the need for wrappers could be avoided altogether.

7.4 Hierarchical components and scheduling

In the existing LTE downlink transmit system, the modulate, scramble and MIMO encoder blocks are wrapped in the SCH and BCH/CCH channels by hierarchical blocks called `sch_modulation` and `cch_modulation`, while a slightly different combination of blocks

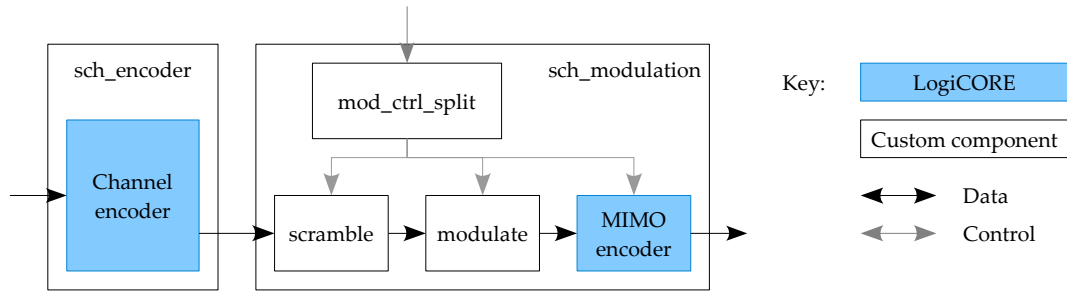


Figure 7.3: SCH encoder and modulation chain.

in the CFICH/HICH is wrapped by the `ich_modulation` component. One of the reasons that blocks are combined in this manner is that they consume control packets at the same rate, and aggregating these packets can reduce the overheads associated with separate transmission through DMA infrastructure. One of the functions of the hierarchical SCH, CCH and ICH blocks is therefore to deaggregate a shared modulation control packet into the control fields required for each of its subcomponents.

7.4.1 Input language

While the actor model allows hierarchical composition of actors, it assumes that computation is performed only at the leaf levels of the hierarchy. This assumption is also made in the OpenDF framework: CAL is used only to describe leaf-level actors, and a separate language called NL is used to describe their coordination [121]. To represent the hierarchical modulation blocks in the NL language, it is therefore necessary to create a new actor to perform the deaggregation of control packets that was previously handled in the hierarchical blocks. This component (called `mod_ctrl_split`) is then included in each of the hierarchical modulation blocks; Figure 7.3 shows how this is done in the SCH chain (together with the encoder block), and Listing 7.1 shows how it is represented in NL.

Listing 7.1: NL representation of SCH modulation hierarchical block.

```
package xlte.dl_tx;
import entity ip.mimo_encoder_v2_0;

network sch_modulation () din, ctrl ==> dout :

entities
    m_scramble = scramble();
    m_modulate = modulate();
    m_mimo = mimo_encoder_v2_0(...);
    m_mod_ctrl_split = mod_ctrl_split();

structure
    din --> m_scramble.din;
    m_scramble.dout --> m_modulate.din;
    m_modulate.dout --> m_mimo.din;
    m_mimo.dout --> dout;

    ctrl --> m_mod_ctrl_split.sch_mod_in;
    m_mod_ctrl_split.scramble_out --> m_scramble.ctrl;
    m_mod_ctrl_split.modulate_out --> m_modulate.ctrl;
    m_mod_ctrl_split.mimo_wrap_out --> m_mimo.ctrl;

end
```

7.4.2 XMODEL code generation

Hierarchical components are represented in IP-XACT in the same manner as leaf-level components, and can be distinguished in the code generation process by their absence of dataflow actions in the component description and presence of one or more subcomponent interconnections in a referenced design description. The code generated for hierarchical components is broadly similar to that for leaf components, but with a few differences. Firstly, the subcomponents in the IP-XACT design description are declared in the .h file and initialized in a constructor in the .cc file. Secondly, in hierarchical components, the FIFOs on input and output ports are unnecessary: input tokens are simply pushed downwards to the input ports of the appropriate subcomponents, and output token pops are requested from the subcomponents' output ports. Thirdly, the `fire_all` function, instead of calling generated action functions, coordinates the movement of data between subcomponents.

The execution of an actor-oriented system on a sequential processor proceeds through a sequence of actor firings, and these are coordinated by passing the execution context to each component in the system. This may be done in a number of ways: for example, a scheduler could simply iterate through a list of all of the components in the system. In the XMODEL framework, scheduling code must be written manually in each hierarchical component to coordinate the interactions of each of its subcomponents, and often proceeds according to an algorithm such as the following:

```

done ← false;
while !done do
  done ← true;
  for i in subcomponent interconnections do
    if !i.sourceport.empty() then
      done ← false;
      token ← i.sourceport.pop();
      i.targetport.push(token);
    end if
  end for
end while

```

To reduce system implementation time, this scheduling code is generated automatically for all hierarchical XMODEL components in the dataflow LTE systems. As an example of this algorithm's realisation in C++, the following code would be generated for the data path of the downlink transmit modulation component.


```

void sch_modulation::process()
{
    bool done = false;

    while (!done)
    {
        done = true;

        if (!m_scramble.dout_empty())
        {
            done = false;
            scramble_dout_t token;
            m_scramble.dout_pop(token);
            m_modulate.din_push(token);
        }

        if (!m_modulate.dout_empty())
        {
            done = false;
            modulate_dout_t token;
            m_modulate.dout_pop(token);
            m_mimo.din_push(token);
        }

        [...]
    }
}

```

Since the `empty` and `push` functions call the `fire_all` function internally, the execution context is passed to all of the subcomponents of each hierarchical component, and processing may be initiated on the whole system by calling the `fire_all` function on the top-level component.

After every actor firing that involves the production of output tokens, those output tokens are pushed into an output FIFO and the execution context returns to the parent component. The parent component then pops this token from the source component's output FIFO, pushes it into the target component's input FIFO, and passes the execution context to the target.

Note that in the algorithm listed above, the data types communicated by the subcomponents of a hierarchical component must be available, and this requires the subcomponents' IP-XACT definitions to be available before the generation of hierarchical components. Thus, a code generation ordering is imposed upon the system.

7.4.3 Enforcing correct code generation order

In order to generate code for hierarchical components, IP-XACT definitions of their sub-components must be available. There are three reasons for this:

- IP-XACT design interconnections do not specify the direction of data flow, instead listing a pair of interfaces that are interconnected in an arbitrary direction. In XMODEL hierarchical nodes, directionality information is needed so that code can be generated which pops tokens from one port and pushes tokens into the other port. It could be assumed that the first listed interface in the interconnection is the master and the second is the sink, but this is not stated in the IP-XACT standard and would cause problems in integrating external IP which does not conform to this assumption. A safer approach is to determine interface direction at code generation time by extracting master/slave interface directions from the IP-XACT component descriptions of the subcomponents.
- IP-XACT design interconnections do not specify the data type transmitted over the interconnection. This information is required for hierarchical component scheduling code (although it may not be required if the scheduling code were implemented differently).
- IP-XACT string literals are not quoted, while C++ string literals are quoted. This means that if a hierarchical component sets a parameter on a subcomponent, C++ code to set this parameter must be generated either with quotes or without, depending on the type of the parameter, and parameter types are associated with subcomponent IP-XACT component descriptions which must be loaded.

This means that IP-XACT descriptions of subcomponents must be generated before the C++ code for their encapsulating hierarchical components. One way to solve this is to add the entire IP-XACT library to the makefile dependency list for generated C++ files, ensuring that all IP-XACT files are created before any C++ code is generated. However, this means that any change to any component requires a full regeneration and recompilation of the C++ code for every generated component in the system. Instead, dependencies are generated on-the-fly.

7.5 Action guards

Certain actions should only be fireable if some property holds of the input tokens, or of the internal state of the component. These properties are known as *guards*, and they can depend either on static (or compile-time configurable) properties of an actor, or on dynamic properties of the received tokens in the action or the component state.

An example of an LTE downlink component requiring guards is the Resource Mapper. This is a hierarchical component consisting of a subcomponent for each LTE channel, and each subcomponent uses control data to calculate the location of the data in that channel in the resource grid. Control data is provided in two packets: one that is specific to the channel, and one that is common to all channels. Each resource mapper subcomponent outputs a packet of data to be written to that location in the subframe memory.

In the case of the PDSCH, PDCCH, PCFICH and PHICH channels, the data to be written is received from the appropriate modulation chain and a variable number of transport blocks may be written in each subframe of data, with the last transport block signalled with a `last` field in the resource mapper control packet. If there are no transport blocks for a particular channel in a subframe, then the `last` flag is set together with a `null` flag. Other channels such as the PSCH generate data on-the-fly and do not require a data input, however the PSCH channel has transport blocks only in subframes 0 and 5. The RS (reference symbol) channel always has a single transport block in every subframe. Only when a resource mapper subcomponent has a non-null transport block will it output data to the subframe memory controller.

This dynamic behaviour can be described in CAL using two guarded actions for each of the SCH, CCH, ICH and PSCH resource mapper subcomponents. The SCH, CCH and ICH guards test the channel-specific resource mapper control packet to determine whether the TB is null, while the PSCH guard tests the common control packet to determine whether the subframe is 0 or 5. The SCH resource mapper actions are represented in CAL as follows:

```

process: action din:      [a],
                      ctrl_sch: [sch],
                      ctrl_com: [com],
                      rbmap:    [rbmap]
==> dout:      [process(a, sch, com, rbmap)]
                      repeat get_tb_size(a, sch, com, rbmap)

guard
  sch.null_tb != 1 || sch.last_tb != 1
end

process_sch_ctrl_only: action ctrl_sch: [sch]
                      ==> dout:      [process_null_tb(sch)]

guard
  sch.null_tb == 1 && sch.last_tb == 1
end

```

Once the guards are represented in CAL, they must be converted into a form that is representable in the IP-XACT intermediate representation, and the standard way to represent expressions in IP-XACT is to use XPath. Since the guards in this example refer to tokens that are received as inputs to the action, a dynamic dependency must be stored in XPath as described in Chapter 5 and as shown in the following listing:

```

<xilinx-dsp:guards>
  <xilinx-dsp:value xilinx-dsp:resolve="runtime-dependent"
    xilinx-dsp:dependency=
      "spirit:decode(field(id('SCH'),'null_tb')) != 1
      or spirit:decode(field(id('SCH'),'last_tb')) != 1"/>
</xilinx-dsp:guards>

```

The runtime-dependent resolution mode causes some difficulty, since these XPath expressions are stored in metadata and cannot be evaluated directly at run-time. To solve this, C++ code is generated from XPath expressions and this becomes part of the executable code defining the component. However, since no XPath-to-C++ conversion is supported by XSLT or the base TT2 package, XPath expressions are parsed in a TT2 template, with the parse tree used to generate equivalent expressions in C++. Details of this process are omitted, but using this technique allows the following output code to be produced, which peeks at the token on the `ctrl_sch` port and then tests it against the guard condition. If the guard condition evaluates to `FALSE`, the action function returns `FALSE` and the action does not fire.

```
sch_ctrl_packet sch;  
m_auto_fifo_ctrl_sch.dout_peek(sch);  
if (!(sch.get_null_tb() != 1 || sch.get_last_tb() != 1)) return false;
```

7.6 Remaining components

Most of the important features of the design flow have now been described through consideration of a number of the upstream components in the LTE downlink transmit system. For completeness, the dataflow abstractions of the remaining components in that system will now be described.

7.6.1 Subframe memory controller

The input data interfaces of the memory controller receive data from each of the resource mapper subcomponents until there is no more data to be read on any of its inputs (i.e. a last flag has been set on all inputs). When data is received, it is loaded into the memory and once a null flag for an interface is received, the controller admits no new data on that interface. Once all interfaces have received a null flag, a read is requested for all data in the memory and it is read out into the next component in the downlink processing chain.

The subframe memory controller has been implemented with independent actions for each data input and an action for data output. Each of the input actions has a guard which prevents further firings when the last flag has been set, and upon each firing the component will update a flag stating whether or not all of the data has been received for that channel.

7.6.2 OFDM

The OFDM component consists of an XFFT core and a number of additional processing components including zero padding, scaling, descaling, cyclic prefix addition and control deaggregation. Each of these blocks is abstracted to a dataflow representation which is used to generate XMODEL classes as described previously.

7.7 Integration into Vivado tool suite

As an alternative to NL, hierarchical systems may be constructed using the Vivado IP Integrator tool, which was released after I had already used NL to describe the model. However, since IP Integrator represents the future evolution of Xilinx system design tools more closely, it is desirable to demonstrate that it can also be used to construct software models. Since both IP Integrator and NL are used as minimal coordination languages, and since both can be used to create IP-XACT design descriptions, design input from IP Integrator may be added to the flow with minimal difficulty.

To do this, it is necessary for components to be integrated into the Vivado IP Catalog; this requires Xilinx-specific metadata to be added, which is normally added using the Vivado IP Packager. However, the IP Packager does not allow arbitrary XML code to be added to component descriptions as are required in the data types and behavioural metadata, and instead requires all additional information to be added as component or bus interface parameters.

There are two possible solutions to this: the first is to determine what metadata the IP Packager adds, and add it outside of the Packager using an XSLT transform, and I have implemented this in a transform called `addXilinxInfo.xsl`. The IP Catalog can then be pointed to the repository of generated IP-XACT definitions using a Tcl command.

The second solution, implemented by Andrew Dow in Xilinx Scotland, is to add the information using the mechanisms that are supported by the Packager. In this solution, Tcl functions are implemented that can be used to create hierarchical data types, which are then flattened into a string which can be stored as a bus interface parameter in IP-XACT. So, for example, the type in Listing 5.1 would be represented as:

```

<spirit:busInterface>
  <spirit:name>S_AXIS_DIN</spirit:name>
  <spirit:parameter>
    <spirit:name>DEF_TDATATYPE</spirit:name>
    <spirit:value>
      spirit:id="BUSIFPARAM_VALUE.S_AXIS_DIN.DEF_TDATATYPE">datatype
      {real {bitwidth {value 5} unsigned fixed {fractwidth {value
      4}}}}</spirit:value>
    </spirit:parameter>
  </spirit:busInterface>

```

While this representation has some disadvantages in comparison to an XML representation of the same information, such as its need for a different parser and its incompatibility with XML processing languages such as XPath and XSLT, it does specify the data type information explicitly in an unambiguous manner.

7.8 Test methodology

Once a complete system has been constructed, it must be tested to ensure that the dataflow abstraction has not introduced any errors. In the original system, test vectors were produced at the input and output interfaces of the system and at various locations within the system. In the generated system, the input vectors were read in and used to stimulate the model, and test vectors were automatically generated on every component interface in the system. By comparing the relevant vectors in the original system and the generated system, it was possible to determine that the generated system was functionally correct.

In order to do this, additional dataflow components were required which could read and write test vectors. Data types for these vectors were available, together with type-polymorphic XMODEL data source and sink components. To allow these to be integrated into an IP Integrator system, dataflow abstractions were created which generated an IP-XACT description, in the same way as the other components in the system.

To represent the type-polymorphic nature of these components, type parameters were required in the CAL descriptions:

```

actor poly_file_source [T] (String txt_file) ==> T dout :

  ifstream m_ifstream;

  action ==> dout: [read_token()]
    guard not_empty()
  end

end

```

Implementations of the `read_token` and `not_empty` functions were adapted from the original source and sink components and included in the user-defined function file. The `ifstream` declaration was used to maintain the state of the file access in a C++ object, and in future work, it would be desirable to represent this in a platform-independent manner.

7.9 Results

Both LTE systems were reconstructed to be functionally equivalent to the original models, with no observable impact on performance since the execution time of the bit-accurate software models dominates over the execution time of the system-level interconnection code. Thus, success is measured by the ability to describe the systems more concisely, and one metric that may be used to measure this is the number of lines of code that are used to describe the system. This is not an ideal measure, since it does not take account of issues such as coding style, but does at least indicate the degree of success.

The original XMODEL systems for the downlink transmit and uplink receive were constructed from approximately 16,000 lines of C++ code. In the automatically generated models, 3,000 lines of high-level domain-specific language (CAL, ASN.1 and RMAP) were used to generate 18,000 lines of C++ code, and 4,000 lines of code from the original model were reused as action functions.

Thus, the total code requirement was reduced by more than 50%, and the requirement for C++ code was reduced by around 75%.

7.10 Discussion

While the presented results appear to be impressive and should apply generically across a variety of system designs, it is believed that if some improvements were made to XMODEL that factor out commonly-used component design patterns, systems could be designed with a reduced amount of C++ code. In turn, the results from generating this smaller amount of code automatically would be somewhat less impressive, but the ratio of input code to generated code is likely to remain appealing.

There are a number of other limitations of the proposed approach. Firstly, since the generated code uses the XMODEL modelling infrastructure rather than an industry-standard set of libraries such as SystemC, it cannot be used outside of Xilinx. The production of a SystemC back-end would therefore be a useful goal in future work. Secondly, the selection of high-level languages that are used as inputs to the flow presents a large amount of syntactic variation, and an improved flow would integrate better with new Xilinx tools such as the Vivado IP Packager. This would require the Packager to be adapted such that it produces all of the dataflow and data type metadata required by the tools described in this chapter. Thirdly, comparisons with other software modelling and code generation approaches are required: for example, it may be preferable for IP-XACT descriptions to be interpreted by a generic software model rather than for the model to be generated from the IP-XACT description. Finally, Figure 4.3 proposes that components could be mapped to different architectures as required, but this is not implemented.

7.11 Conclusions

This chapter has demonstrated that two LTE signal processing systems may be represented using high-level descriptions, that these descriptions may be converted to a metadata format expressed in XML, and that these metadata descriptions may be used to generate a large amount of C++ code from a relatively small amount of input code whilst preserving the functional correctness of the model: reductions of 75% in the amount of C++ code were observed.

Chapter 8

Conclusion

This chapter concludes the thesis by summarising the contributions described in previous chapters, and outlines some future research directions.

The aim of the thesis was to tackle the issue of creating FPGA systems more efficiently, so that the potential benefits of reconfigurable logic over microprocessor-based processing platforms can be realised. Challenges in FPGA system design include the need to reuse existing IP cores, explore the design space, model the system before implementation, design components at a high level of abstraction, maintain good quality of results, and maintain interoperability with other tools.

To address these issues, a toolflow was presented which takes account of the trend in industry towards the capture of component and system metadata in increasing levels of detail, and demonstrates that the schema for this metadata can be used as a metamodel for an intermediate representation in a model-driven engineering design flow.

This required the metadata to be extended to include descriptions of components' data types and high-level behaviour. Contributions in these extensions included the capturing of:

- structure, array and complex types allowing the description of the streaming packets that are communicated over component ports, together with encodings described as individual bit offsets or as strides in an array;

- timing constraints and latency between the different interfaces in a component, and the representation of interface blocking using dataflow actions; and
- dynamic component characteristics.

It was stated that this additional information can be used to infer appropriate logic to connect components that would otherwise have to be written manually. An example of this was demonstrated in the context of multi-dimensional array types, and it was shown how reorder blocks could be synthesized either from existing SID cores or from automatically-generated C code. It was then demonstrated that depending on the way in which the dimensions are propagated around the system, a variety of buffering scenarios could be produced and a process for determining the most efficient scenario was described. In large systems, this process significantly improves the ability for a designer to assess the implementation options in the arrangement of array dimensions, and to select the most efficient solution.

Finally, a toolflow was demonstrated which converts high-level, abstract component and system descriptions, via the proposed metadata schema, to an executable software model of the system. Integration is demonstrated with existing Xilinx tools and processes such as the Vivado IP Catalog and IP Integrator, and the XMODEL simulation framework. By designing the system at a high level, the number of lines of input code was reduced by more than 50%, and around 75% of the original C++ code could be automatically generated.

8.1 Limitations and future work

This section will summarise the limitations of each of the main contributions of the thesis.

In the process of determining an appropriate metamodel, IP core features are encountered that require a sufficiently complex description that their representation in metadata may not be appropriate. These include IPv4 options that are more simply described by the imperative code that processes them than by a static metadata description, and FIR advanced channel sequences, for which an expression determining the location of a par-

ticular element in a particular channel is not obvious. Further work is required to determine whether it is possible to represent these features in metadata, and furthermore to determine whether this remains beneficial in light of any additional complexity that the extended representation brings.

In using the metadata to perform buffering optimisations, complications are encountered in the inevitability of naming conflicts between the array dimensions of different cores. While these can be handled manually by the user, no demonstration of this process exists and on a broader level, a demonstration of the whole process in the context of a tool such as Vivado IP Integrator is required to prove the utility of the approach.

Finally, in the use of the metamodel in a code generation flow, limitations were encountered in the choice of an XMODEL backend rather than the industry-standard SystemC and the heterogeneity of the high-level language inputs, and questions were raised as to whether generating software code was preferable to interpretation of IP-XACT descriptions. Also, the desirability of user-specified component mapping to allow hardware-in-the-loop testing was noted: whilst work had begun on this topic, the results were not sufficiently mature to be described in this thesis.

Of the limitations described above, the most pressing is the need to demonstrate interoperability with existing Xilinx tools. To this end, a high priority for future work is for the Vivado IP Packager to generate data type and dataflow component metadata for all of the cores in the Vivado IP Catalog, for the Vivado IP Integrator to use array metadata to generate reorder buffers automatically and determine minimal buffering scenarios, and for automatically generated buffer code to be passed to Vivado HLS for fast implementation.

Other high-priority goals for future work are to use behaviour metadata to generate wrappers allowing the automatic integration of latency-sensitive cores in contexts where data latencies cannot be guaranteed, and to build IP core models around a unified API, demonstrating that this allows automatic integration with no need for wrappers to be created manually.

Finally, some other targets for future work are to capture FIR advanced channel sequences and IP option headers in the data type schema, or determine conclusively that no

solution to these challenges is possible, and to demonstrate that heterogeneous systems can be built from a single metadata system description, with a mapping file determining which components are implemented on which processing platforms in the system.

Bibliography

- [1] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobbs’s Journal*, vol. 30, no. 3, pp. 202–210, 2005. [20](#)
- [2] J. Koomey, “Estimating total power consumption by servers in the US and the world,” Tech. Rep., 2007. [20](#)
- [3] “TOP500 List – June 2012.” [Online]. Available: <http://top500.org/list/2012/06/100> [20](#)
- [4] W.-c. Feng and K. W. Cameron, “The Green500 List: Encouraging Sustainable Supercomputing,” *IEEE Computer*, vol. 40, no. 12, pp. 50–55, 2007. [20](#)
- [5] A. Lingamneni, K. Krishna, C. Enz, R. M. Karp, and C. Piguet, “Algorithmic Methodologies for Ultra-efficient Inexact Architectures for Sustaining Technology Scaling,” in *Proc. 9th conference on Computing Frontiers (CF)*, 2012, pp. 3–12. [21](#)
- [6] V. Betz and S. Brown, “FPGA Challenges and Opportunities at 40 nm and Beyond,” in *Proc. 19th International Conference on Field Programmable Logic and Applications*, 2009. [21](#)
- [7] T. Erjavec, “Introducing the Xilinx Targeted Design Platform: Fulfilling the Programmable Imperative,” Xilinx, Tech. Rep., 2009. [21](#), [26](#)
- [8] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Sys.*, vol. 26, no. 2, pp. 203–215, 2007. [22](#)

- [9] M. Wirthlin, B. Nelson, B. Hutchings, P. Athanas, and S. Bohnner, "Future Field Programmable Gate Array (FPGA) Design Methodologies and Tool Flows," Air Force Research Laboratory, Wright-Patterson Air Force Base, OH, Tech. Rep. July, 2008. [22](#)
- [10] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The Promise of High-Performance Reconfigurable Computing," *IEEE Computer*, vol. 41, no. 2, pp. 69–76, May 2008. [22](#)
- [11] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and Power of Cache-Based Reconfigurable Computing," in *Proc. 36th annual International Symposium on Computer Architecture*, 2009, pp. 395–405. [22](#)
- [12] X. Tian and K. Benkrid, "High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU," *ACM Trans. Reconfigurable Technology and Sys. (TRETs)*, 2010. [22](#)
- [13] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978. [22](#)
- [14] S. Singh, "Computing without processors," *Communications of the ACM*, vol. 54, no. 8, pp. 46–54, Aug. 2011. [23](#)
- [15] "MicroBlaze Processor Reference Guide," Xilinx, Technical Manual, 2012. [23](#)
- [16] "Nios II Processor Reference Handbook," Altera, Tech. Rep., 2011. [23](#)
- [17] "Virtex-5 Family Overview," Xilinx, Product Guide, 2009. [23](#)
- [18] "Zynq-7000 All Programmable SoC Overview," Xilinx, Product Guide, 2012. [23](#)
- [19] "Strategic Considerations for Emerging SoC FPGAs," Altera, Tech. Rep., 2011. [23](#)
- [20] D. Wall, "Limits of instruction-level parallelism," *Architectural Support for Programming Languages and Operating Sys.*, vol. 19, no. 2, 1991. [24](#)

- [21] D. Patterson, "The trouble with multi-core," *IEEE Spectrum*, vol. 47, no. 7, pp. 28–32, 2010. 24
- [22] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006. 25
- [23] "Catapult C," Calypto, 2013, computer program. [Online]. Available: <http://calypto.com/en/products/catapult/overview> 25
- [24] "Vivado Design Suite User Guide," Xilinx, User Guide, 2012. 25, 52, 64
- [25] T. Hill, "AccelDSP Synthesis Tool: Floating-Point to Fixed-Point Conversion of MATLAB Algorithms Targeting FPGAs," Xilinx, White Paper, 2006. 25
- [26] "Low Power Hybrid Computing for Efficient Software Acceleration," Mitrionics, AB, White Paper, 2008. 25
- [27] "Handel-C Language Reference Manual," Agility Design Solutions, Technical Manual, 1999. 25
- [28] "FIR Compiler v6.3 LogiCORE Product Specification," Xilinx, Data Sheet, 2011. 26
- [29] A. Allan, D. Edenfeld, W. H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, "2001 Technology Roadmap for Semiconductors," *IEEE Computer*, vol. 35, no. 1, pp. 42–53, 2002. 26
- [30] B. Nelson, M. Wirthlin, B. Hutchings, P. Athanas, and S. Bohnert, "Design productivity for configurable computing," in *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2008. 26
- [31] *Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, IEEE Std. 1685, 2009. 27, 46
- [32] T. P. Perry and R. L. Walke, "Determining efficient buffering for multi-dimensional datastream applications," U.S. Patent 8,365,109, January 29, 2013. 28
- [33] T. P. Perry, R. L. Walke, and K. Benkrid, "An extensible code generation framework for heterogeneous architectures based on IP-XACT," in *Proc. 7th Southern Conference on Programmable Logic (SPL)*, 2011. 28

- [34] T. P. Perry, R. L. Walke, R. Payne, S. Petko, and K. Benkrid, "IP-XACT extensions for IP interoperability guarantees and software model generation," in *Proc. 22nd International Conference on Field Programmable Logic and Applications*, 2012. [28](#)
- [35] M. Rumney, G. Jue, M. Stambaugh, B. Zarlingo, R. Becker, B. Irvine, S. Fraser, P. Cain, R. Yonezawa, H. Yanagawa, M. Obara, E. W. Koo, P. Kangru, C. Van Woerkom, M. Yokoyama, B. Ying, P. Goldsack, P. Gupta, Z. Lovell, J.-P. Gregoire, P. Jones, M. Leung, K. F. Tsang, S. Singh, S. Charlton, V. Ratnakar, D. Sabharwal, and N. Das, *LTE and the Evolution to 4G Wireless*. Wiley-Blackwell, 2009. [29](#)
- [36] C. Lim, T. Yoo, B. Clerckx, B. Lee, and B. Shim, "Recent Trend of Multiuser MIMO in LTE-Advanced," *IEEE Communications Magazine*, vol. 51, no. 3, pp. 127–135, 2013. [32](#)
- [37] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Sys.*, vol. 19, no. 12, pp. 1523–1543, 2000. [44](#), [54](#), [69](#)
- [38] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design & Test of Comput.*, vol. 18, no. 6, pp. 23–33, 2001. [44](#)
- [39] E. A. Lee, S. Neuendorffer, and M. Wirthlin, "Actor-oriented design of embedded hardware and software systems," *J. Circuits, Sys. and Comput.*, vol. 12, no. 3, pp. 231–260, 2003. [44](#), [45](#), [60](#)
- [40] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. A. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and Others, "OpenFPGA CoreLib core library interoperability effort," *Parallel Computing*, vol. 34, no. 4-5, pp. 231–244, 2008. [45](#)
- [41] M. Birnbaum, F. Microelectronics, C. C. Johnson, and I. Corp, "VSIA quality metrics for IP and SoC," in *Proc. IEEE 2001 2nd International Symposium on Quality Electronic Design*, 2001, pp. 279–283. [45](#)

- [42] R. Bergamaschi, W. R. Lee, D. Richardson, S. Bhattacharya, M. Muhlada, R. Wagner, A. Weiner, and F. White, "Coral - Automating the Design of Systems-On-Chip Using Cores," in *Proc. IEEE 2000 Custom Integrated Circuits Conference*, 2000, pp. 109–112. [45](#)
- [43] P. E. McKechnie, "Validation and Verification of the Interconnection of Hardware Intellectual Property Blocks for FPGA-based Packet Processing Systems," EngD thesis, Institute for System Level Integration, Universities of Edinburgh, Glasgow, Heriot-Watt and Strathclyde, 2010. [45](#), [57](#)
- [44] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in *Proc. 35th annual Design Automation Conference*. ACM, 1998, pp. 8–13. [45](#)
- [45] V. D'Silva, S. Ramesh, and A. Sowmya, "Bridge over troubled wrappers: Automated interface synthesis," *Proc. 17th International Conference on VLSI Design*, pp. 189–194, 2004. [45](#)
- [46] W. Kruijtzter, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. de Paoli, and E. Vaumorin, "Industrial IP integration flows based on IP-XACT standards," in *Design, Automation and Test in Europe*. IEEE, Mar. 2008, pp. 32–37. [46](#)
- [47] A. Arnesen, N. Rollins, and M. Wirthlin, "A multi-layered XML schema and design tool for reusing and integrating FPGA IP," in *Proc. 19th International Conference on Field Programmable Logic and Applications*, 2009, pp. 472–475. [49](#)
- [48] A. Arnesen, D. Gibelyou, and M. Wirthlin, "IP-XACT Extensions for Interface Synthesis in Reconfigurable Computing," Tech. Rep., 2010. [49](#), [103](#)
- [49] N. Rollins, A. Arnesen, and M. Wirthlin, "An XML schema for representing reusable IP cores for reconfigurable computing," in *Proc. 2008 National Aerospace and Electronics Conference (NAECON)*, Jul. 2008, pp. 190–197. [50](#)
- [50] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin, "Increasing Design Productivity Through Core

- Reuse, Meta-Data Encapsulation and Synthesis,” in *Proc. 20th International Conference on Field Programmable Logic and Applications*, 2010, pp. 538–543. [50](#)
- [51] *SystemC*, OSCI Std., Rev. 2.0, 2002. [50](#)
- [52] B. Hutchings and B. Nelson, “Using general-purpose programming languages for FPGA design,” *Proc. 37th Design Automation Conference (DAC)*, pp. 561–566, 2000. [52](#)
- [53] B. Bond, K. Hammil, L. Litchev, and S. Singh, “FPGA Circuit Synthesis of Accelerator Data-Parallel Programs,” in *Proc. 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 167–170. [52](#)
- [54] “MaxCompiler White Paper,” Maxeler Technologies, 2011. [52](#), [64](#)
- [55] S. Aditya and V. Kathail, “Algorithmic Synthesis Using PICO,” in *High-Level Synthesis: from Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds. Springer, 2008, ch. 4, p. 55. [53](#)
- [56] J. Noguera, S. Neuendorffer, K. A. Vissers, and C. Dick, “Wireless MIMO Sphere Detector Implemented in FPGA,” *XCell*, pp. 38–45, 2011. [53](#)
- [57] J. Bézivin, “On the unification power of models,” *Software and Sys. Modeling*, vol. 4, no. 2, pp. 171–188, 2005. [53](#), [55](#)
- [58] A. Sangiovanni-Vincentelli, S. K. Shukla, J. Sztipanovits, G. Yang, and D. A. Mathaikutty, “Metamodeling: An Emerging Representation Paradigm for System-Level Design,” *IEEE Design & Test of Comput.*, vol. 26, no. 3, pp. 54–69, May 2009. [54](#)
- [59] T. Mens and P. Van Gorp, “A Taxonomy of Model Transformation,” *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, Mar. 2006. [54](#)
- [60] R. Soley, “Model driven architecture,” Object Management Group, Tech. Rep., 2000. [55](#)
- [61] *Unified Modeling Language*, OMG Std., Rev. 2.4.1, 2011. [Online]. Available: <http://www.uml.org> [55](#)

- [62] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, Jul. 2004. [55](#)
- [63] A. El Mrabti, F. Petrot, and A. Bouchhima, "Extending IP-XACT to support an MDE based approach for SoC design," in *Design, Automation and Test in Europe*, 2009, pp. 586–589. [55](#)
- [64] R. Nane, S. V. Haastregt, T. Stefanov, B. Kienhuis, V. M. Sima, and K. Bertels, "IP-XACT Extensions for Reconfigurable Computing," in *Proc. 22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2011, pp. 215–218. [55](#)
- [65] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, 1992. [56](#)
- [66] S. K. Shukla, F. Doucet, and R. K. Gupta, "Structured Component Composition Frameworks for Embedded System Design," in *Proc. 9th International Conference on High Performance Computing (HiPC)*, 2002, pp. 663–678. [56](#)
- [67] F. Doucet, S. K. Shukla, R. K. Gupta, and M. Otsuka, "An environment for dynamic component composition for efficient co-design," in *Proc. 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 736–743. [56](#)
- [68] F. Doucet, S. K. Shukla, M. Otsuka, and R. K. Gupta, "Balboa: A component-based design environment for system models," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Sys.*, vol. 22, no. 12, pp. 1597–1612, 2003. [56](#)
- [69] D. A. Mathaikutty and S. K. Shukla, "MCF: A Metamodeling-Based Component Composition Framework Composing SystemC IPs for Executable System Models," *IEEE Trans. Very Large Scale Integration (VLSI) Sys.*, vol. 16, no. 7, pp. 792–805, 2008. [56](#), [64](#)
- [70] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A Model-Driven Design Framework for Massively Parallel Embedded Systems," *ACM Trans. Embedded Computing Sys.*, vol. 10, no. 4, pp. 1–36, Nov. 2011. [56](#)

- [71] P. Boulet, "Array-OL Revisited, Multidimensional Intensive Signal Processing Specification," INRIA, Tech. Rep. January, 2007. [56](#), [91](#)
- [72] V. Aggarwal, G. Stitt, A. George, and C. Yoon, "SCF: A Framework for Task-Level Coordination in Reconfigurable, Heterogeneous Systems," *ACM Trans. Reconfigurable Technology and Sys. (TRETs)*, vol. 5, no. 2, 2012. [56](#), [64](#)
- [73] J. Hwang, B. Milne, N. Shirazi, and J. Stroomer, "System level tools for DSP in FPGAs," in *Proc. 11th International Conference on Field Programmable Logic and Applications*, 2001, pp. 534–543. [57](#), [64](#)
- [74] "EDK Concepts, Tools and Techniques: A Hands-On Guide to Effective Embedded System Design," Xilinx, User Guide, 2011. [57](#)
- [75] "Vivado IP Integrator: accelerated time to IP creation and integration," Xilinx, White Paper, 2013. [57](#), [64](#)
- [76] P. G. Whiting and R. S. Pascoe, "A history of data-flow languages," *IEEE Annals of the History of Computing*, vol. 16, no. 4, pp. 38–59, 1994. [58](#)
- [77] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proc. IFIP Congress*, 1974, pp. 471–475. [58](#)
- [78] E. A. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, 1995. [58](#), [59](#)
- [79] E. A. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987. [58](#)
- [80] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Sig. Proc.*, vol. 44, no. 2, pp. 397–408, 1996. [59](#)
- [81] T. Parks, J. Pino, and E. A. Lee, "A comparison of synchronous and cyclo-static dataflow," in *Proc. 29th Asilomar Conference on Signals, Systems and Computers*, no. October. IEEE Comput. Soc. Press, 1995, pp. 204–210. [59](#)

- [82] J. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, University of California at Berkeley, 1993. 59
- [83] J. T. Buck, E. M. Rd, and M. View, "Static Scheduling and Code Generation from Dynamic Dataflow Graphs With Integer- Valued Control Streams," in *Proc. 28th Asilomar Conference on Signals, Systems and Computers*, 1994, pp. 508–513. 59
- [84] J. Piat, "Interface-based hierarchy for synchronous data-flow graphs," in *IEEE Workshop on Signal Processing Systems*, vol. 1, 2009, pp. 145–150. 59
- [85] P. K. Murthy and E. A. Lee, "Multidimensional Synchronous Dataflow," *IEEE Trans. Sig. Proc.*, vol. 50, no. 7, pp. 2064–2079, 2002. 59, 124
- [86] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003. 60, 64
- [87] P. D. O. Castro, S. Louise, and D. Barthou, "Reducing memory requirements of stream programs by graph transformations," in *Proc. International Conference on High Performance Computing and Simulation (HPCS)*, 2010, pp. 171–180. 60
- [88] "SPW," Synopsys, 2013, computer program. [Online]. Available: <http://www.synopsys.com/Systems/BlockDesign/DigitalSignalProcessing/Pages/Signal-Processing.aspx> 60, 64
- [89] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2009. 60, 64
- [90] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs," *J. Sig. Proc. Sys.*, vol. 63, no. 2, pp. 241–249, Jul. 2011. 60
- [91] J. Eker and J. W. Janneck, "CAL language report," University of California, Berkeley, Tech. Rep., 2003. 60

- [92] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *J. Sig. Proc. Sys.*, vol. 63, no. 2, pp. 251–263, Jul. 2011. [61](#)
- [93] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures," in *Proc. 8th International Workshop on Hardware/Software Codesign*, 2000, pp. 13–17. [61](#)
- [94] H. Nikolov, M. Thompson, T. Stefanov, and A. D. Pimentel, "Daedalus: Toward Composable Multimedia MP-SoC Design," in *Proc. 45th annual Design Automation Conference*, 2008, pp. 574–579. [61](#), [64](#)
- [95] B. Kienhuis, E. Deprettere, and K. A. Vissers, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," in *Proc. 1997 IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 1997, pp. 338–349. [61](#)
- [96] H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-processor System Design with ESPAM," in *Proc. 4th international conference on Hardware/software codesign and system synthesis*, 2006, pp. 211–216. [61](#)
- [97] K. Huang and J. Gu, "Automatic Platform Synthesis and Application Mapping for Multiprocessor Systems-On-Chip," Master's thesis, LIACS - Leiden University, 2005. [61](#)
- [98] A. D. Pimentel, T. Stefanov, H. Nikolov, M. Thompson, S. Polstra, and E. F. Deprettere, "Tool Integration and Interoperability Challenges of a System-Level Design Flow: A Case Study," in *Proc. 8th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2008, pp. 167–176. [61](#), [70](#)
- [99] C. Lattner, "LLVM," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds. lulu.com, 2011, ch. 11. [70](#)
- [100] *Abstract Syntax Notation One (ASN.1)*, ITU-T Recommendation X.680, 2002. [72](#), [85](#)
- [101] "ML605 Hardware User Guide," Xilinx, User Guide, 2012. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf [74](#)

- [102] N. Walsh, A. Milowski, and H. S. Thompson, *XProc: An XML Pipeline Language*, W3C Recommendation, Rev. 11 May 2010, 2010. [Online]. Available: <http://www.w3.org/TR/xproc> 75
- [103] R. M. Stallman and R. McGrath, *GNU Make — A Program for Directing Recompilation*. Free Software Foundation, Cambridge, MA, USA, 1991. 75
- [104] M. Lamb, “Nailgun: Insanely Fast Java,” 2012, computer program. [Online]. Available: <http://www.martiansoftware.com/nailgun> 76
- [105] O. Ben-Kiki, C. Evans, and B. Ingerson, *YAML Ain’t Markup Language (YAML)*, Working Draft, Rev. 1.2, December 2004. 79
- [106] R. Orfali and D. Harkey, *Client/server programming with Java and CORBA*, 2nd ed. Wiley, Mar. 1998. 85
- [107] “Protocol Buffers,” computer program. [Online]. Available: <http://code.google.com/p/protobuf> 85
- [108] M. Slee, A. Agarwal, and M. Kwiatkowski, “Thrift: Scalable Cross-Language Services Implementation,” Facebook, Tech. Rep., 2007. 85
- [109] *ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, ITU-T Recommendation X.690, 2002. 87
- [110] *ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*, ITU-T Recommendation X.691, 2002. 87, 148
- [111] *ASN.1 encoding rules: Specification of Encoding Control Notation (ECN)*, ITU-T Recommendation X.692, 2002. 87
- [112] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message passing interface*. MIT Press, 1999. 91
- [113] E. A. Lee and D. Messerschmitt, “Synchronous Data Flow,” *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. 122
- [114] “SymPy,” software library. [Online]. Available: <http://www.sympy.org> 124

- [115] L. de Moura and N. Bjørner, “Satisfiability modulo theories: an appetizer,” *Formal Methods: Foundations and Applications*, pp. 23–36, 2009. [125](#)
- [116] L. de Moura and N. Bjoerner, “Z3: An Efficient SMT Solver,” in *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, pp. 337–340. [125](#)
- [117] R. Wille, D. Große, M. Soeken, and R. Drechsler, “Using Higher Levels of Abstraction for Solving Optimization Problems by Boolean Satisfiability,” in *International Symposium on VLSI*, 2008, pp. 411–416. [126](#)
- [118] P. Suter, A. Köksal, and V. Kuncak, “Satisfiability modulo recursive programs,” *Static Analysis*, pp. 298–315, 2011. [128](#)
- [119] X. Liu and Y. Zhao, “The Component Interaction Domain: Modeling Event-Driven and Demand- Driven Applications,” in *5th Biennial Ptolemy Miniconference*, 2003. [144](#)
- [120] T. J. Parr and R. W. Quong, “ANTLR: A Predicated-LL(k) Parser Generator,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. [146](#)
- [121] J. W. Janneck, “NL – a Network Language,” Programmable Solutions Group, Xilinx Inc., Tech. Rep., 2007. [157](#)

Appendix A

Publications

AN EXTENSIBLE CODE GENERATION FRAMEWORK FOR HETEROGENEOUS ARCHITECTURES BASED ON IP-XACT

Thomas P. Perry, Richard L. Walke

Xilinx Development Corporation,
Charles Darwin House,
Edinburgh Technopole, Bush Estate,
Milton Bridge, Penicuik, UK
email: Thomas.Perry@slu-institute.ac.uk,
Richard.Walke@xilinx.com

Khaled Benkrid

University of Edinburgh,
School of Engineering,
King's Buildings,
Mayfield Road,
Edinburgh, UK
email: K.Benkrid@ed.ac.uk

ABSTRACT

In this paper, we examine the problem of abstracting the design process for heterogeneous CPU/FPGA systems from the perspective of a group of engineers designing telecommunications systems, and propose a design flow that addresses the constraints imposed in an industrial context whilst striving for maximal compatibility with existing tools and research projects.

We thus present a modular and extensible flow based around the IP-XACT standard, which is gaining support in industry, and link this to a front-end built on the semantics of dataflow process networks and a template-based code generation back-end.

1. INTRODUCTION

With the increasing size of programmable logic devices such as FPGAs, it is becoming increasingly difficult to design complex systems that make full use of the resources that these devices offer. In addition, new architectures such as the Xilinx Extensible Processing Platform (EPP)[1] offer greater opportunities for heterogeneous processing but design methodologies must be adapted to exploit this potential. Thirdly, despite the compelling performance and power advantages of FPGAs over competing technologies in new domains such as High Performance Reconfigurable Computing, uptake has been stifled by the steep learning curve and low-level nature of existing tools[2].

In response to these difficulties, attempts have been made to raise the level of abstraction of the FPGA design process by creating tools to automate the generation of low-level, architecture-specific code, but these tools have met

some resistance in the marketplace.

While the parallel complexity of FPGAs undoubtedly complicates the task of designing intuitive tools, we do not believe that it is a simple failure to address this complexity that limits the tools' broader adoption. Instead, we believe that failures of uptake may in most cases be explained by one or more of the following reasons:

choice of abstraction abstractions may be chosen that do not provide a good model for the problem at hand, and thus high-level description and code generation techniques become unwieldy.

applicability the abstraction is sound but the implementation flow is a 'point solution', insufficiently general to be of widespread use.

lock-in the tool allows no recourse to a prior flow, necessary in the event that it fails to demonstrate the anticipated benefits.

aggregation of flows the tool conflates a number of aspects of the design process that could remain separate and be addressed using separate mechanisms, and thus the tool is rejected on its weaknesses rather than embraced on its strengths.

The first of these reasons is commonly stated in the academic community (e.g. [3]), and we encounter the second and third in informal discussions with experienced engineers. The fourth derives from the principle of *separation of concerns*, stated by Dijkstra[4] and finds application in, for example, the Unix philosophy of 'do one thing and do it well'. It has also been invoked in the domain of system-level design (e.g. [5]) and thus its implications will now be discussed.

In the design of any highly parallel system implemented on a platform such as an FPGA, there are three separable

This work was supported by the UK Engineering and Physical Sciences Research Council and Scottish Enterprise through an FHPCA EngD studentship, and by an Industrial Fellowship of the Royal Commission for the Exhibition of 1851.

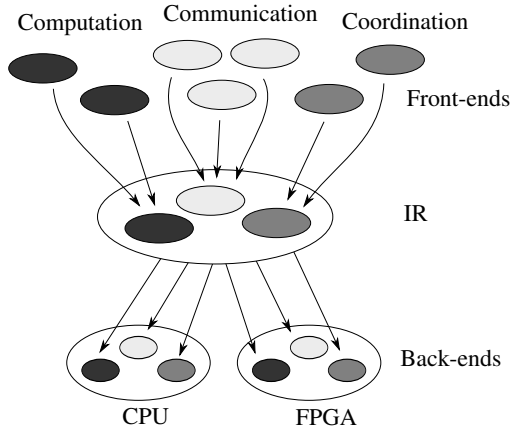


Fig. 1. Separating concerns in a design flow.

aspects of the problem that must be defined: the *computation* performed by the processing nodes, stated as sequences of arithmetic, logical and other operations, the *communication* between the external interfaces of connected nodes, requiring characterisation of data type serialization and low-level data transmission processes, and the *coordination* of the nodes, comprising the (possibly dynamic) topology of the network that connects them and the interrelation of their patterns of execution and data exchange. In other words, we can characterise a parallel node by what it does, what it says, and when and to whom it talks. C-to-gates technology, for example, is marketed on the basis of the familiarity of its entry method, but the C input only serves to address the requirement for computational descriptions and the communication and coordination of the generated components must be described by proprietary methods.

Each of these three aspects may also be separated into front-end and back-end flows, connected via the use of some intermediate representation (IR) format, as shown in Figure 1. This division is fundamental to the operation of compilers in general, but finds even broader appeal in compilation flows for heterogeneous architectures where multiple back-ends are required, and additionally in flows for homogeneous architectures where multiple front-ends may be desired (providing, for example, both textual and graphical entry methods). If the IR is standardised, the risk of lock-in is reduced. If the IR is also extensible, an ecosystem of front-ends and back-ends targeting specific domains may be constructed to extend the applicability of the original flow.

Through this division into three aspects and two levels, the overall task of defining a system design methodology has been decomposed into six sub-tasks that may be tackled largely independently. In the following sections, we deliberately set aside the computational aspect of the problem,

since (a) there already exist adequate (though architecture-specific) languages for computational descriptions in micro-processors and FPGA fabric, (b) it has been difficult thus far to demonstrate improvements over these languages in industrial contexts, and (c) there already exists a large body of ongoing research with such a demonstration as its aim. Instead, we attempt to address the underspecification of communication and coordination, with initial efforts focusing on the 3GPP LTE physical layer systems described in the following section, but in a manner that we intend subsequently to be more broadly applicable.

2. ANALYSIS OF INTENDED APPLICATIONS

The mobile communications industry is currently benefiting from significant growth, and will in coming years be fueled by the adoption of the 3GPP Long Term Evolution (LTE) standard[6].

The engineering team with which we are working produces FPGA cores and software simulation models written in VHDL and C++ that implement individual processing stages of the LTE physical layer (PHY), and include these furthermore in full LTE reference designs and software system simulations. It can be observed, however, that the cores may often be considered as systems since they incorporate a number of sub-cores, and since systems are also typically integrated into larger designs by customers, the notions of core and system used to characterise and market these products may simply be reconciled for our purposes into one of hierarchical composition.

Our objective in the first instance is to automatically generate code for an LTE PHY that would otherwise have been written manually, and since this target is characteristic of a broader range of intended applications (to be elaborated in Section 2.3), it is possible to derive an appropriate abstraction through analysis of the existing code. Characterising the software and hardware implementations separately, we see the following:

The software models make use of a custom C++ framework which provides a generic component class, and this class must be extended for each category of required component to provide input/output functions and one or more processing functions. A model thus consists of a number of instances of these extended classes, often connected by FIFO queues, that communicate using function calls. Data elements are composed in structures and arrays, and for the sake of efficiency may be communicated using pointers. The execution of each component is driven by a schedule manually specified in an encapsulating component, and this schedule is static where the flow rates of the subcomponents' packets are apparent, and dynamic otherwise.

The reference design consists, similarly, of a number of component instances often separated by FIFOs, in this

case communicating most commonly using streaming interfaces. Data packets are represented as VHDL records, but are serialized before they are transmitted between components. Since this communication happens only when both the sender asserts a valid signal and the receiver asserts a ready signal, no external scheduling is required.

In characterising the similarities between these systems, it may be observed that the communication style in each system fits the paradigm of asynchronous message passing, and with the addition of some semantic rules that simplify but do not overconstrain the character of these systems, it becomes possible to use the language of *dataflow process networks* to describe them at a higher level of abstraction.

2.1. Dataflow process networks

A dataflow process network is a collection of *actors* with input and output ports that each send or receive tokens via point-to-point, unidirectional token *arcs*[7]. Also associated with each actor is a list of *actions*, each defining expressions that transform tokens on input ports to tokens on output ports, and through the repeated *firing* of these actions upon arrival of sufficient input tokens, an actor executes these expressions and generates a stream of output tokens which are sent to other actors. The topology of the actors' interconnection is captured by a dataflow graph, and by considering the structure of this graph together with the properties of the actors, it is possible to reason about the communication patterns and thereby perform automatic optimisations such as static scheduling that improve the performance of the system[8].

The dataflow model as described above states no particular requirements on the specification of data types, so an abstraction for these must be found elsewhere.

2.2. Data types and encodings

The communicational aspect of system design requires characterisation of two separate sub-aspects: firstly, abstraction of the hierarchical structure of data types, and secondly, abstraction of the manner in which these types are encoded in packets. According to the commonly observed principle of separating content (the type structure) from form (the encoding), these sub-aspects should be specified separately. The first of these may be used to generate data type definitions in architecture-specific languages, and the second may be used to generate appropriate serialization and deserialization routines that are required to transmit tokens of these types. In Section 4.2, a possible approach based on existing technology is suggested.

2.3. Other applications

By reducing the LTE PHY to its dataflow fundamentals, it becomes apparent that this target is merely one of a more general class of applications, characterised by high volumes of data and frequent communication, for which a more abstract view is beneficial.

While control-heavy or highly dynamic applications are unlikely to benefit from a dataflow style of coordination, there remain a number of targets in other forms of signal processing and in certain high performance computing (HPC) applications, such as iterative grid calculations, that may be amenable to a dataflow model. One example for which the dataflow abstraction is appropriate is MPEG video coding[9].

Having proposed a set of applications and a unifying abstract model, it must now be demonstrated that a high-level representation of these applications may be converted to a standard but extensible intermediate representation format. In the sections below, we propose and justify the use of IP-XACT for this purpose.

3. IP-XACT AS AN INTERMEDIATE REPRESENTATION

IP-XACT is an industry-standard XML schema that is being adopted by many organisations in the FPGA industry to describe metadata about their cores and systems in a manner that is independent of the implementation language.

A number of top-level object descriptions are permitted, including component descriptions which store information pertaining to individual components, and design descriptions which store a list of components and their connections to each other and to the encapsulating component. Hierarchy is represented by reference in a component description to a design description listing a number of sub-components.

Extensibility is possible through the use of IP-XACT *vendor extension* tags under which custom XML fragments may be stored, and schemas for this information may be specified externally. Defining such schemas has been the focus of research at CHREC[10], and Neely et al. make similar use of IP-XACT extensions in their own framework[11].

Together with a suitable set of vendor extensions, the IP-XACT schema may thus be used to record the information required in the coordinational aspect of system design.

3.1. Required IP-XACT extensions

The standard IP-XACT schema does not allow the specification of dataflow information such as firing rules and token rates for components, but recent work at CHREC has demonstrated a set of vendor extensions suitable for this task[12] and we implement a subset of their features in our own extension schema. We have an additional requirement,

however, for an expressive data type schema that may be used to describe the packet structures used in LTE control and data streams.

A close match may be found in the work of Risso and Baldi in the form of their NetPDL schema[13], however as this is a microprocessor-oriented solution, the syntax for describing bit-aligned fields as found in FPGA systems involves bit masks rather than the preferable IP-XACT approach of lengths and offsets, and the notion of packet *form* is implicit: packets are structured based on their contents rather than from an explicit specification or selectable rule set.

Thus, no single XML schema meets all of our requirements, and we have defined a bespoke IP-XACT vendor extension schema that we call ‘XCI’. Using this schema, it is possible to define:

- Dataflow token rates and firing rules for components, and references to ‘actions’ to be performed in C++ or VHDL.
- Type definitions comprising leaf-level *units* (with customizable bit widths, offsets, binary point presence and placement, and numerical bounds), and hierarchical *arrays* (with or without a defined maximum size) and *structures* consisting of sub-fields.

With the type extensions described above, it is possible to generate type definitions in both C and VHDL that allow efficient execution and communication. For example, boolean values specified using this schema might be implemented in C using an 8-bit `char`, while in VHDL they would be implemented as a single-bit value. Interface functions would then be generated automatically that allow communication of this boolean value between software and hardware.

4. FRONT-ENDS

Having characterised the target problems and specified an internal representation, we now attempt to find high-level design entry methods that adequately capture the information required to describe the class of systems we aim to generate.

There are various commercial tools available for high-level design of dataflow systems, including GEDAE and the Synopsys SPW tool, and we are currently investigating the generation of IP-XACT descriptions from some of these tools.

As a basis for research beyond the timescales of commercial evaluation licenses, however, the openness of the OpenDF dataflow toolset provides compelling advantages.

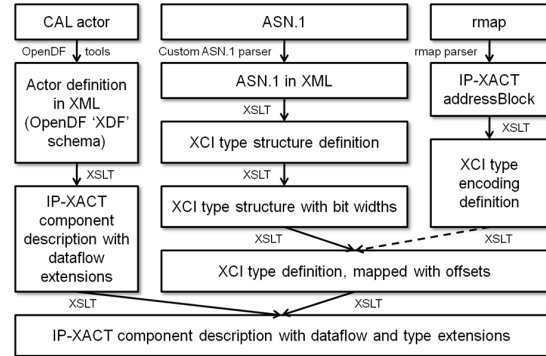


Fig. 2. IP-XACT generation flow for leaf-level components.

4.1. OpenDF

The OpenDF project[14] is a set of tools for the construction of networks of dataflow actors. Two high-level languages are specified: CAL, which is used to describe the interfaces and operation of individual dataflow actors[15], and NL, which acts as a minimal coordination language suitable for describing the components’ linear and hierarchical composition. Descriptions in these languages may be run in a supplied simulator or compiled so as to be run natively on various processing devices: for example, a CAL-to-HDL back-end is provided such that CAL actors may be instantiated on FPGAs[16].

Included in the compilation flow performed by the tools in the OpenDF suite is a stage in which dataflow component descriptions in CAL and coordinational descriptions in NL are converted into XML form. Selective conversion of the coordinational aspects of this XML output to our extended IP-XACT schema is therefore feasible, and we have achieved this using XSLT[17].

Since we explicitly avoid in this work the issue of defining computational descriptions of components, our tools undertake no transformation of CAL output expressions (the token transformations executed when an action fires) into IP-XACT; they simply translate an NL description into an IP-XACT design and encapsulating component, and a CAL description into an IP-XACT leaf component that references an output expression specified in native code (C++ or VHDL).

4.2. High-level type definitions

The OpenDF tools do not specify a mechanism for defining data types, and instead delegate this responsibility to external tools and flows.

High-level description of data types is already done in Xilinx LTE systems using a bespoke ‘rmap’ (register map)

language, which is used to define the structure of non-hierarchical control register maps. A Perl script is then used to generate low-level driver and HAL code in C and VHDL from this representation.

The rmap language cannot be used to describe hierarchical types, however, and register descriptions are only characterised by their size and offset. In order to generate type metadata that includes all of the features listed in Section 3.1, another high-level representation must be found.

One high-level language that is suited to this task is ASN.1 (Abstract Syntax Notation One)[18], which may be used to specify the hierarchical structure of data types and the characteristics of leaf-level fields (for example: integer, enumeration, plain bit field). Inputs in this format are parsed into an XML parse tree format, and these are converted using XSLT to our custom ‘XCI’ schema (see Figure 2).

Encodings may be defined manually or generated automatically. Using the manual approach, encodings are specified using the rmap format, and these are then converted to XCI format and merged with the specification derived from ASN.1. Alternatively, the encoding is generated using XSLT transformations which assign bit widths and offsets for each field automatically. The generated XCI representation is then added as an IP-XACT vendor extension to the main IP-XACT component description.

5. CODE GENERATION FROM IP-XACT

Generating an IP-XACT definition of a system is useful in itself, since this definition may be provided to other tools that require an IP-XACT input, but an additional benefit arises in the ability to automatically generate output code for multiple targets and in multiple languages from a single specification.

At the core level, labour-saving developments could include generation of C interfaces to the C++ core models, VHDL entity declarations, data type serialization/deserialization routines, SystemVerilog DPI functions for testing, core testbenches and wrappers for interoperability with MATLAB. At the system level, we aim to fully automate the generation of hierarchical interconnection logic and scheduling functions. Thus, our code-generation requirements are manifold and our chosen approach must be extensible by future users.

5.1. Template approach

Code generation techniques may be broadly classified into those that store output code fragments inline with the remainder of the generator source, and those that store this code separately in templates. The essential difference between these approaches may once again be distilled down to separation of concerns, in this case separation of presentation and application logic, or separation of form from the mechanism with which form is applied to content.

In order to encourage adoption of any code generation framework, it is desirable firstly for the output code to be easily readable, and furthermore, for it to look as much as possible like the code that would have been written manually. These properties prevent lock-in by allowing later recourse to manual editing of the generated files, but they also require a greater degree of flexibility in presentation.

To provide this flexibility, we build our code generation strategy around a template processing engine with code templates that are manipulable by the user and populated upon demand from an IP-XACT definition. We use the Perl Template Toolkit[19] for this purpose, since it is widely used and understood.

To demonstrate the intuitive nature of the template approach, we pass the IP-XACT output from the previous example to the Perl Template Toolkit engine together with the following template, which specifies how the contents of the `componentInstances` tag in an IP-XACT design description should be processed.

```
// Subcomponents
[% FOREACH ci IN design.componentInstances %]
[% ci.componentRef.name %]& [% ci.instanceName %];
[% END %]
```

The output from this flow is a pair of C++ subcomponent declarations, as follows.

```
// Subcomponents
xlte_scramble& m_scramble;
xlte_modulate& m_modulate;
```

Using this approach, we generate C++ subclasses of a pre-existing component class and add component-specific communication and coordination code. The implementations of computational functions (output expressions) that are produced using an external process are stored in separate files and are incorporated into the build at link-time.

6. FUTURE WORK

The aim in writing the templates that we have produced so far is to recreate the existing software model structure as accurately as possible.

This model does not fully conform to the dataflow process network model of computation, since components’ output ports are demand-driven rather than data-driven. Written this way, it is possible to avoid associating target actors with output ports since an encapsulating component is responsible for all of the token transfers in and out of a subcomponent.

Generating fully dataflow code would allow the automatic generation of static schedules for hierarchical nodes, so this will be attempted in further work.

In a similar manner to the generation of C++ from IP-XACT using templates, we aim to create VHDL templates

to allow an implementation on FPGA fabric. With the addition of a mechanism for message passing between software and hardware components, the C++ and VHDL code generation flows may be combined to allow implementation on a heterogeneous platform such as the Xilinx EPP.

7. CONCLUSION

We have proposed the use of IP-XACT with appropriate extensions as an intermediate representation in a compilation flow from high-level, domain-restricted communication and coordination languages to low-level, architecture-specific languages. In order to demonstrate this flow, we have implemented conversions from a dataflow front-end to IP-XACT and provided a template-based back-end that allows user-extensible code generation. This is currently being tested through generation of C++ code for Xilinx LTE software models, but VHDL templates will be produced in future work in order to automate the low-level communication and coordination code for designs implemented on FPGA fabric.

We have also defined an XML schema for the description of data types used in Xilinx cores and systems, and tested this by writing descriptions in this format for the data types used by a number of existing cores and system components. In order to allow high-level entry, we have implemented a compilation flow that generates these descriptions using ASN.1 specifications as input.

8. REFERENCES

- [1] K. DeHaven, "Extensible Processing Platform," Xilinx, Tech. Rep. April, 2010.
- [2] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin, and C. Kitchen, "An overview of FPGAs and FPGA programming; Initial experiences at Daresbury," Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Tech. Rep. November, 2006.
- [3] M. Wirthlin, B. Nelson, B. Hutchings, P. Athanas, and S. Bohner, "Future Field Programmable Gate Array (FPGA) Design Methodologies and Tool Flows," Air Force Research Laboratory, Wright-Patterson Air Force Base, OH, Tech. Rep. July, 2008.
- [4] E. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, Mar. 1982, vol. 12, no. 3, pp. 60–66.
- [5] W. Cesário, G. Nicolescu, L. Gauthier, D. Lyonnard, and A. Jerraya, "Colif: A design representation for application-specific multiprocessor SOCs," *IEEE Design & Test of Comput.*, vol. 18, no. 5, pp. 8–20, 2001.
- [6] D. McQueen, "The momentum behind LTE adoption," *IEEE Communications Magazine*, vol. 47, no. 2, pp. 44–45, 2009.
- [7] E. A. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, 1995.
- [8] E. A. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.
- [9] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *J. Sig. Proc. Sys.*, July 2009.
- [10] A. Arnesen, N. Rollins, and M. Wirthlin, "A multi-layered XML schema and design tool for reusing and integrating FPGA IP," in *Field Programmable Logic and Applications*, 2009, pp. 472–475.
- [11] C. Neely, G. Brebner, and W. Shang, "ShapeUp: A High-Level Design Approach to Simplify Module Interconnection on FPGAs," in *18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 141–148.
- [12] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin, "Increasing Design Productivity Through Core Reuse, Meta-Data Encapsulation and Synthesis," in *Field Programmable Logic and Applications*, 2010, pp. 538–543.
- [13] F. Risso and M. Baldi, "NetPDL: An extensible XML-based language for packet header description," *Computer Networks*, vol. 50, no. 5, pp. 688–706, Apr. 2006.
- [14] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2009.
- [15] J. Eker and J. W. Janneck, "CAL language report," University of California, Berkeley, Tech. Rep., 2003.
- [16] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs," *J. Sig. Proc. Sys.*, July 2009.
- [17] J. Clark, "XSL transformations (XSLT)—version 1.0," World Wide Web Consortium, Tech. Rep., 1999. [Online]. Available: <http://www.w3.org/TR/xslt>
- [18] "Recommendation X.680: Abstract Syntax Notation One (ASN.1)," ITU-T, Tech. Rep., 2002.
- [19] D. Chamberlain, D. Cross, and A. Wardley, *Perl Template Toolkit*. O'Reilly & Associates, Inc., 2003.

IP-XACT EXTENSIONS FOR IP INTEROPERABILITY GUARANTEES AND SOFTWARE MODEL GENERATION

Thomas P. Perry, Richard L. Walke, Rob Payne, Stefan Petko*

Xilinx Development Corporation,
Charles Darwin House,
Edinburgh Technopole, Bush Estate,
Milton Bridge, Penicuik, UK
email: firstname.lastname@xilinx.com

Khaled Benkrid

University of Edinburgh,
School of Engineering,
King's Buildings,
Mayfield Road, Edinburgh, UK
email: K.Benkrid@ed.ac.uk

ABSTRACT

This paper presents a set of novel metadata extensions that are used to specify the interfaces on Xilinx IP cores and their software models under a uniform data model which allows enhanced design rule checking in the system design process. We also present a suite of tools which can be used to generate executable software simulation models of complete systems from their specifications under that data model. These tools may be used stand-alone, or may be used to extend the capabilities of the Vivado IP Integrator tool that [as of the intended publication date] has recently been released by Xilinx.

Our toolflow has been used successfully to generate software simulation models for two 3GPP Long Term Evolution (LTE) physical layer systems: uplink receive and downlink transmit.

1. INTRODUCTION

The capacity of FPGA devices is continually increasing, and there is a concomitant rise in the complexity of the systems designed for them. A commonly accepted mechanism for dealing with this complexity is to use IP cores, but for an IP-based design flow to be of use, the task of connecting cores together must be significantly more cost-effective than designing those blocks from scratch.

To address the issue of IP usability, Xilinx has released Vivado IP Integrator. This is a new tool that allows stitching of abstract, pre-configured IP blocks (for example, with no bus widths chosen) using either a graphical editor or a Tcl-based textual interface, with intelligent parameter propagation used to assign configurations to those blocks before generation of HDL to link them together.

*This work was supported by the UK Engineering and Physical Sciences Research Council and Scottish Enterprise through FHPCA EngD studentships, and by an Industrial Fellowship of the Royal Commission for the Exhibition of 1851.

One aspect of IP usability is the provision of appropriate design rule checks (DRCs) in a system-level design tool, and thus one of the design goals of IP Integrator is for the tool to provide assistance in determining which interface connections are compatible and which should be disallowed. When incompatible cores are connected together in the tool, it should flag an error or propose an intelligent solution to the problem, which might involve generation of additional infrastructure to enable connections in situations where cores are 'almost' compatible. However, this process requires detailed information about IP cores to be provided.

The first of the two novel contributions made by this paper is to define a set of metadata extensions that may be layered on top of the existing IP core metadata in order to extend the design rule checking capabilities of the IP Integrator tool and to permit automatic data type coercion on connections between similar interfaces. These extensions focus particularly on the data types communicated by IP cores.

The second contribution tackles another issue of design productivity, which is that for a system of significant size and complexity, development of that system does not always start with the interconnection of cores in the HDL domain. Instead, a software model is produced which connects bit-accurate models of those cores and integrates bespoke software components which perform additional data movement and manipulation functions. Due to the higher-level input representation, greater levels of architectural exploration are permitted than when stitching cores in HDL. The simulation model created in this process is used to generate test vectors at a variety of points in the system and HDL components are instantiated or created to replicate the functionality of each of the software blocks.

However, recreating the system in two different languages is laborious. Our key realisation, introduced in previous work[1], is that much of the information required to generate these systems has already been captured in component metadata for the purpose of ensuring interoperability,

and that a software simulation model or a hardware implementation can be generated from this metadata together with a system design description which is also stored in metadata. In this paper, we demonstrate this with the automatic generation of LTE software models, and find that the only significant requirement beyond the metadata is a C++ function to be executed in a ‘dataflow action’, and in the case of many Xilinx IP cores (LogiCOREs), this function already exists in the form of a bit-accurate software model.

Thus we conclude that with the current metadata, plus some extensions that we describe, plus dataflow actions specified in C++, the automatic generation of a software system model is possible.

The remainder of this paper is organised as follows. In Section 2 we discuss useful background to the problem. In Sections 3 and 4 we present our metadata extensions. In Section 5 we show how this metadata can be used to generate a software system simulation model. In Section 6 we state our current and future avenues for research, and in Section 7 we conclude the paper.

2. BACKGROUND AND RELATED WORK

We use two Xilinx LTE baseband telecommunications systems as the focus of our research: a downlink transmit (DL TX) system and an uplink receive (UL RX) system.

There are a number of situations in the Xilinx LTE systems where one component outputs a multidimensional array of data and the core to which it is connected requires the same array structure but with the dimensions in a different order. Figure 1 shows one example of this in the context of the UL RX system: the Channel Estimator v1.1 LogiCORE groups data elements by ‘codeword’ then by ‘antenna’, whereas the MIMO Decoder v2.1 groups elements by ‘antenna’ then by ‘codeword’[2]. If these cores are connected directly, an incorrect sequence of data transfers will occur: a design environment should either prevent this by refusing to connect the components, or propose a solution.

All Xilinx LogiCOREs have an associated metadata description in the IP-XACT schema, which is gaining broad acceptance as a standard schema in which to represent metadata about electronic components and systems, such as bus interface types.

In the IP-XACT schema, a number of top-level object descriptions are permitted, including component descriptions, which store information pertaining to individual components, and design descriptions which can be used to represent hierarchical designs consisting of those components[3].

The information in IP-XACT component descriptions can be used to determine compatibility between two connected components at the level of bus interfaces. Previous work, for example in the Coral tool[4], deals with compatibility at the interface level. Since Xilinx is standardising

on the AMBA AXI interface, the compatibility issue within the domain of AXI-compatible cores may be considered at a higher level of abstraction: the problem becomes one of ensuring data type and dataflow compatibility.

At various points in the IP-XACT schema, vendor-specific extensions may be included to extend the description. Previously, other authors have proposed extensions to IP-XACT for this purpose[5], and we take a similar approach but focus in greater depth on data type descriptions.

Other authors have produced tools which generate software models from a high-level description[6]. However, the introduction of novel high-level languages is difficult to motivate in industry, so in contrast we base our approach on standard IP-XACT representations of components and systems that already exist.

3. DATA TYPE METADATA

Our approach is influenced by interface definition languages (IDLs) such as ASN.1[7] and Thrift[8]. None of these tools can be used to specify the interfaces of FPGA IP, since they do not allow fine-grained control of data type encodings. Also, in contrast, we do not specify a front-end input language and instead focus on the data model and its representation in XML.

Our schema allows data types to be associated with elements such as bus interfaces and register fields in an IP-XACT component description. These may be specified inline, as follows:

```
<spirit:busInterface>
  <spirit:vendorExtensions>
    <x:dataType>
      ...
    </x:dataType>
  </spirit:vendorExtensions>
</spirit:busInterface>
```

Since data types are often shared between components, we also allow for libraries of types to be created and referenced in IP-XACT component descriptions:

```
<spirit:busInterface>
  <spirit:vendorExtensions>
    <x:dataTypeRef
      spirit:vendor="example.com"
      spirit:library="lte"
      spirit:name="resource_block"
      spirit:version="1.0" />
    </spirit:vendorExtensions>
</spirit:busInterface>
```

In components with a configurable data type, the type may be specified using an elaboration-time configurable parameter, which allows polymorphic components to be described:

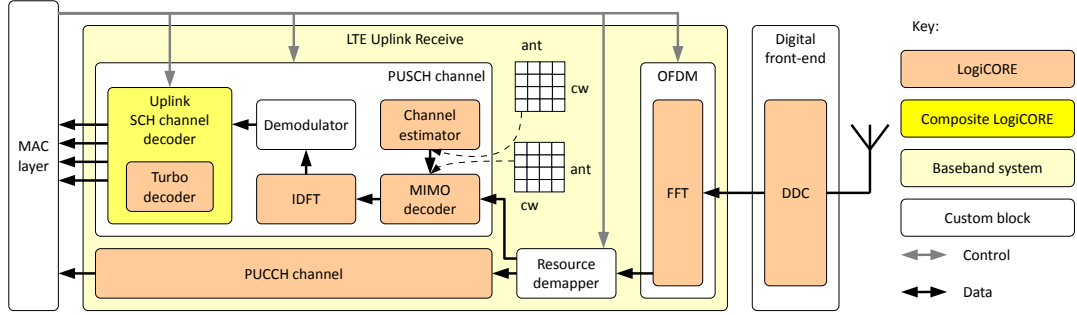


Fig. 1. The Xilinx LTE UL RX system, showing a data type interoperability issue between the channel estimator and MIMO decoder blocks due to different orderings of array dimensions.

```

<spirit:component>
  <spirit:busInterface>
    <spirit:vendorExtensions>
      <x:configurableDataTypeRef
        spirit:resolve="depends"
        spirit:format="string"
        spirit:dependency
          ="id('PARAM_VALUE.OPERAND_TYPE')"/>
      </spirit:vendorExtensions>
    </spirit:busInterface>
    <spirit:parameter>
      <spirit:name>operand_type</spirit:name>
      <spirit:value
        spirit:format="string"
        spirit:id="PARAM_VALUE.OPERAND_TYPE" />
    </spirit:parameter>
  </spirit:component>

```

The schema permits a hierarchy of types consisting of structures and multi-dimensional arrays to be described. Structures and arrays both contain `spirit:field` elements as defined in the base IP-XACT schema, extended to include an interpretation of the constituent bits which may be a boolean, integer, fixed-point value or floating-point value. Complex values are also permitted which can hold two integer, fixed-point or floating-point values.

In the base IP-XACT schema, `bitWidth` and `bitOffset` tags are associated with each field in a register. We extend the interpretation of register fields in the base IP-XACT schema such that they can be included in streaming packets such as those that are transmitted over a streaming interface.

The position of the fields within these packets may be specified using the base IP-XACT `bitWidth` and `bitOffset` tags, but we reinterpret the `bitOffset` tag to indicate the offset from the start of the packet, rather than from the start of a data word within that packet. This allows packets to be described in a manner that is independent of any particular word size, since this may be confused with the width of a particular interface over which they are transmitted. The intended interpretation is that values in the abstract

TDM: false, NANT: 2					TDM: true, NANT: 4				
64	48	32	16	0	64	48	32	16	0
t ₀	Q0A2	I0A2	Q0A1	I0A1	t ₀	I0A4	I0A3	I0A2	I0A1
t ₁	Q1A2	I1A2	Q1A1	I1A1	t ₁	Q0A4	Q0A3	Q0A2	Q0A1
t ₂	Q2A2	I2A2	Q2A1	I2A1	t ₂	I1A4	I1A3	I1A2	I1A1
t ₃	Q3A2	I3A2	Q3A1	I3A1	t ₃	Q1A4	Q1A3	Q1A2	Q1A1
t _n					t _n				

Fig. 2. DUC/DDC Compiler data format in two modes: no TDM and 2 antennas; TDM and 4 antennas.

type are mapped from least-significant to most-significant bit across successive data beats transmitted across the interface, and this allows the automatic inference of blocks to bridge interfaces with different widths to be handled as an orthogonal issue to the contents of the data streams.

For fields contained in an array or a complex value, we introduce a 'stride' tag. The stride specifies the offset between successive elements of an array, or between the real and imaginary values in a complex value.

The concept of array strides is particularly useful in the DUC/DDC Compiler v2.0 LogiCORE[2], which is a configurable digital up-converter or digital down-converter that operates on parallel streams of complex data to or from a configurable number of radio antennas. The real and imaginary parts of each complex value may be communicated in parallel, or they may be sent in a time-division multiplexed (TDM) form according to a core parameter, as shown in Figure 2.

In attempting to capture both data encoding possibilities in the same metadata description, it is desirable to preserve the same abstract type (a multidimensional array of complex

```

<x:datatype>
  <x:param name="TDM"/>
  <x:param name="NANT"/>
  <x:param name="D_WIDTH"/>
  <x:array>
    <x:name>antennas</x:name>
    <x:size dependency="$NANT"/>
    <x:stride dependency="if ($TDM)
      then $D_WIDTH
      else $D_WIDTH * 2"/>
  <x:datatype>
    <x:complex>
      <x:real>
        <x:bitWidth dependency="$D_WIDTH"/>
      </x:real>
      <x:realInLSBs/>
      <x:stride dependency="if ($TDM)
        then $D_WIDTH * $NANT
        else $D_WIDTH"/>
    </x:complex>
  </x:datatype>
</x:array>
</x:datatype>

```

Fig. 3. DUC/DDC Compiler data format expressed in XML metadata. In parallel mode, the complex value has a stride of one element-width, but in TDM mode the stride is the number of antennas multiplied by the element width. In parallel mode, the antenna array has a stride of two element-widths, but in TDM mode the stride is one element width.

values) and layer a configurable description of the type encoding on top of it. This allows for high-level (e.g. software) interfaces based only upon complex values to be presented to the user, avoiding the need to present different interfaces for different encoding configurations of the core. This can be achieved using array strides, as shown in Figure 3.

By using strides in this way, the behaviour of the data interfaces on the DUC/DDC Compiler can be captured in a single metadata description.

4. DATAFLOW METADATA

The second group of metadata extensions deals with the dataflow properties of IP cores and software components. We use dataflow metadata for two purposes: firstly, to describe the operation of the cores to enable correct integration; secondly, to unify IP cores and their bit-accurate software models under a single model of computation.

Most Xilinx DSP LogiCOREs require some form of control information, and operate in one of a number of modes. In the first mode, control information is processed asynchronously to the data stream, and changes take effect some time after the receipt of the control packet. In the second, there is a fixed dependency between the number of control packets and the number of data packets received: if, for example, a control packet fails to arrive, the data interface

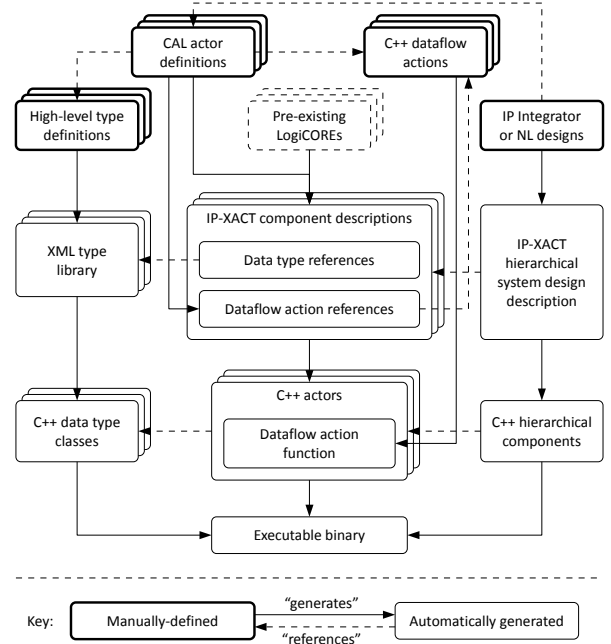


Fig. 4. Software generation flow.

will block until the control packet arrives. Cores of this type are essentially synchronous dataflow (SDF) actors. In the third mode, the relative rates of control and data are variable, depending, for example, on data communicated in the control packet. These may be characterised as dynamic dataflow (DDF) actors.

Examples of the first case are the XFFT v8.0 and FIR Compiler v6.3 LogiCOREs, which have no pre-defined rate relationship between control and data[2]. An example of the second case is the LTE MIMO Decoder LogiCORE. An example of the third case is the LTE PUCCH LogiCORE.

5. GENERATING A LIBRARY OF SOFTWARE COMPONENTS

The need for software modelling has already been described, and the first step we take is to construct a library consisting of three main classes of software components: wrapped IP core models, bespoke components, and polymorphic test vector sources and sinks.

These components may be imported into the Xilinx IP Catalog, allowing IP Integrator to construct software systems in addition to the HDL systems that it can already create from the cores in the Xilinx IP Catalog.

The process is shown in Figure 4 and explained in the following sections.

5.1. Creating data type descriptions

XML data type descriptions are created from a high-level language. For this purpose, we have extended a simple domain-specific language called RMAP that is used internally in Xilinx, but the toolflow is agnostic with regard to the particular input language used.

5.2. Creating IP-XACT component descriptions with dataflow extensions

A dataflow description of the component is created. We use CAL[9], but as in the data type step, the flow is agnostic to the input language.

To create new components, these CAL descriptions are converted to an IP-XACT component description with dataflow extensions and references to the previously-defined data types.

To add dataflow information to existing cores in the IP Catalog, we create a CAL description with the same name as the core and extend the IP-XACT description for that component with the generated dataflow metadata from the CAL description.

5.3. Creating C++ action functions

C++ functions are written to specify the computation to be performed when firing rules specified in the dataflow metadata are satisfied.

Many existing IP blocks have bit-accurate software models, but since their APIs vary slightly they cannot be integrated automatically. To handle this situation, we write small wrappers which map function names referenced in the CAL description to the function names provided by the IP model API.

5.4. Generating C++ code

C++ classes are generated for the data types, leaf-level components, and hierarchical components.

The data type classes contain data members, accessor and mutator methods, and encoding and decoding functions. The encoding functions create a byte stream using the data in the class members, and the decoding functions read data from a byte stream and populate the class members. A type encoding function generated from the XML representation is shown in Figure 5.

Component classes are generated from extended IP-XACT component descriptions. For leaf-level components, the manually-written C++ functions are copied into the generated file. For hierarchical components, a function is generated which schedules data movement between its subcomponents.

Test vector monitor points are generated on every component interface, which use the data type encoding functions

```
void cch_resmap_ctrl_packet::v_append_packet
(xuint32_packet& p) const
{
    size_t s;

    s = p.size();
    p.resize(s + 2);
    p[s] |= (get_last_tb    () & (1 << 1) - 1);
    p[s] |= (get_null_tb    () & (1 << 1) - 1) << 1;
    p[s] |= (get_bch_cch    () & (1 << 1) - 1) << 2;
    p[s] |= (get_start_cce  () & (1 << 7) - 1) << 8;
    p[s] |= (get_cch_format () & (1 << 2) - 1) << 16;
    p[s] |= (get_frame_mod4 () & (1 << 2) - 1) << 24;
    p[s + 1] |= (get_cch_scale () & (1 << 12) - 1);
}
```

Fig. 5. C++ data type encoding function.

to output a byte stream to a test vector. The data in this vector is in the format used by the IP cores and HDL components, allowing testing of hardware blocks against their software model equivalents.

5.5. Compiling C++ code and importing into IP Catalog

The C++ code is compiled into static objects, ready to be linked into a binary that is generated when a system is stitched together.

In order for the components to be imported in the Xilinx IP Catalog, they require IP-XACT files, but we have already generated these. The new or extended IP-XACT component descriptions are added to the IP Catalog with a simple Tcl command.

5.6. Generating software system models

We provide the ability to stitch systems together using a variety of coordination languages, which are all converted to an IP-XACT design description. The NL language is one option, which is provided alongside CAL in the OpenDF toolkit[10]. Another option is Xilinx IP Integrator, which outputs IP-XACT design descriptions natively.

Figure 6 shows an LTE system constructed in IP Integrator, with test vector sources and sinks connected to the inputs and outputs of the system. Each source and sink is configured with a test vector filename and a data type, such that the data in the vector is decoded or encoded correctly.

In its standard use model, IP Integrator generates HDL code to stitch IP cores together. Since the tool exports an IP-XACT design description for any systems it is used to create, the code generation possibilities can be extended in various ways.

In this paper, we stitch together the software components generated as described in Section 5 by generating C++ code from IP-XACT design descriptions.

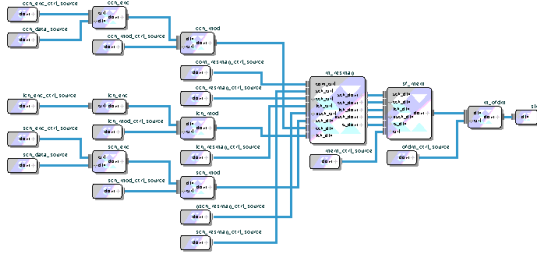


Fig. 6. Screenshot from IP Integrator showing hierarchical blocks in the LTE downlink transmit system connected together with test vector sources and sinks.

This C++ code transfers tokens between its subcomponents by executing the following algorithm:

```

done ← false;
while !done do
  done ← true;
  for i in subcomponent interconnections do
    if !i.sourceport is empty then
      done ← false;
      tokentype ← i.sourceport.type;
      tokentoken ← i.sourceport.pop();
      i.targetport.push(tokentoken);
    end if
  end for
end while

```

6. ONGOING AND FUTURE WORK

Our ongoing work addresses three extensions to this topic that we hope to publish in the near future.

Firstly, with the ability to create a software model and an HDL system of interconnected IP cores from the same high-level description, an obvious next step is to enable the construction of heterogeneous systems with both software and hardware IP components. This would permit a design flow in which a software model is produced first, and then the software components are gradually replaced with hardware components that are tested ‘in-the-loop’.

Secondly, bespoke software components have been created by generating C++ code from metadata and adding a C++ function to specify computation. A similar process could be applied to generate bespoke HDL components. This could be achieved through manual implementation or through high-level synthesis of software code.

Thirdly, with a sufficiently detailed data type schema it becomes possible to automatically coerce the data transmitted on one interface into the type expected on another. For example, if the data array transmitted by the master interface is the transpose of that expected by the slave, a reorder

buffer can be inferred.

7. CONCLUSION

We have described metadata extensions that allow sophisticated design rule checks to be performed by system-level design environments. These extensions are categorised into data type and dataflow groups. We have also shown that once these extensions are added to component descriptions, only a small amount of software code is required as user input in order for the generation of full software system simulation models to be possible. Finally, we have successfully demonstrated our generation flow in two practical systems, namely the uplink receive and downlink transmit physical layers of 3GPP LTE.

8. REFERENCES

- [1] T. P. Perry, R. L. Walke, and K. Benkrid, “An extensible code generation framework for heterogeneous architectures based on IP-XACT,” in *Proc. 7th Southern Conference on Programmable Logic (SPL)*, 2011.
- [2] “Xilinx IP Product Specifications,” Tech. Rep. [Online]. Available: www.xilinx.com/support
- [3] “IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows,” IEEE, Tech. Rep., 2009.
- [4] R. Bergamaschi, W. R. Lee, D. Richardson, S. Bhattacharya, M. Muhlada, R. Wagner, A. Weiner, and F. White, “Coral - Automating the Design of Systems-On-Chip Using Cores,” in *IEEE 2000 Custom Integrated Circuits Conference*, 2000, pp. 109–112.
- [5] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin, “Increasing Design Productivity Through Core Reuse, Meta-Data Encapsulation and Synthesis,” in *Proc. 20th International Conference on Field Programmable Logic and Applications*, 2010, pp. 538–543.
- [6] F. Doucet, S. K. Shukla, M. Otsuka, and R. K. Gupta, “Balboa: A component-based design environment for system models,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Sys.*, vol. 22, no. 12, pp. 1597–1612, 2003.
- [7] “Recommendation X.680: Abstract Syntax Notation One (ASN.1),” ITU-T, Tech. Rep., 2002.
- [8] M. Slee, A. Agarwal, and M. Kwiatkowski, “Thrift: Scalable Cross-Language Services Implementation,” Facebook, Tech. Rep., 2007.
- [9] J. Eker and J. W. Janneck, “CAL language report,” University of California, Berkeley, Tech. Rep., 2003.
- [10] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, “OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems,” *Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2009.



US008365109B1

(12) **United States Patent**
Perry et al.

(10) **Patent No.:** **US 8,365,109 B1**
(45) **Date of Patent:** **Jan. 29, 2013**

(54) **DETERMINING EFFICIENT BUFFERING
FOR MULTI-DIMENSIONAL DATASTREAM
APPLICATIONS**

(75) Inventors: **Thomas P. Perry**, Edinburgh (GB);
Richard L. Walke, Edinburgh (GB)

(73) Assignee: **Xilinx, Inc.**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this
patent is extended or adjusted under 35
U.S.C. 154(b) by 0 days.

(21) Appl. No.: **13/535,123**

(22) Filed: **Jun. 27, 2012**

(51) **Int. Cl.**
G06F 17/50 (2006.01)

(52) **U.S. Cl.** **716/102**; 716/101; 716/114; 716/139

(58) **Field of Classification Search** 716/101,
716/102, 114, 139

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,799,064	A *	8/1998	Sridhar et al.	379/93.06
6,131,182	A *	10/2000	Beakes et al.	716/105
6,698,002	B2 *	2/2004	Chang et al.	716/106
7,210,111	B1 *	4/2007	Smith et al.	716/108
7,333,514	B2 *	2/2008	Anehem et al.	370/474
2005/0289495	A1 *	12/2005	Raghunandran	716/11
2010/0235803	A1 *	9/2010	Gramark et al.	716/12

OTHER PUBLICATIONS

"Bridge Over Troubled Wrappers: Automated Interface Synthesis",
by ViJay D'silva, S. Ramesh, Arcot Sowmya, @2004, IEEE.*

"A Formal Approach to Interface Synthesis for System-On-Chip
Design", by Vijay D'silva, Arcot Sowmya, Sridevan Parameswaran,
S. Ramesh, Apr. 2003.*

Murthy, Praveen K., et al., "Multidimensional Synchronous
Dataflow", IEEE Transactions on signal Processing, vol. 50, No. 7,
Jul. 2002, pp. 2064-2079.

* cited by examiner

Primary Examiner — Thuan Do

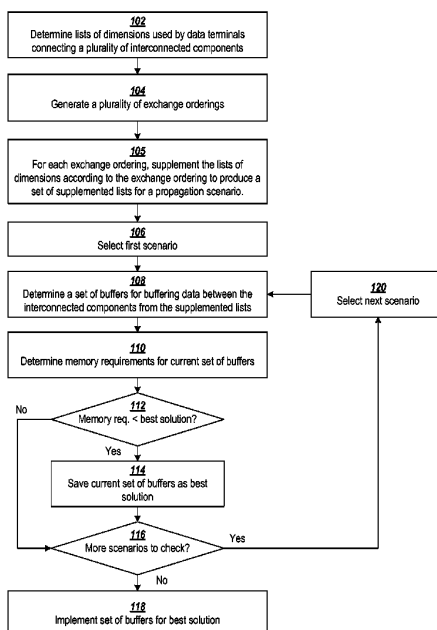
Assistant Examiner — Nha Nguyen

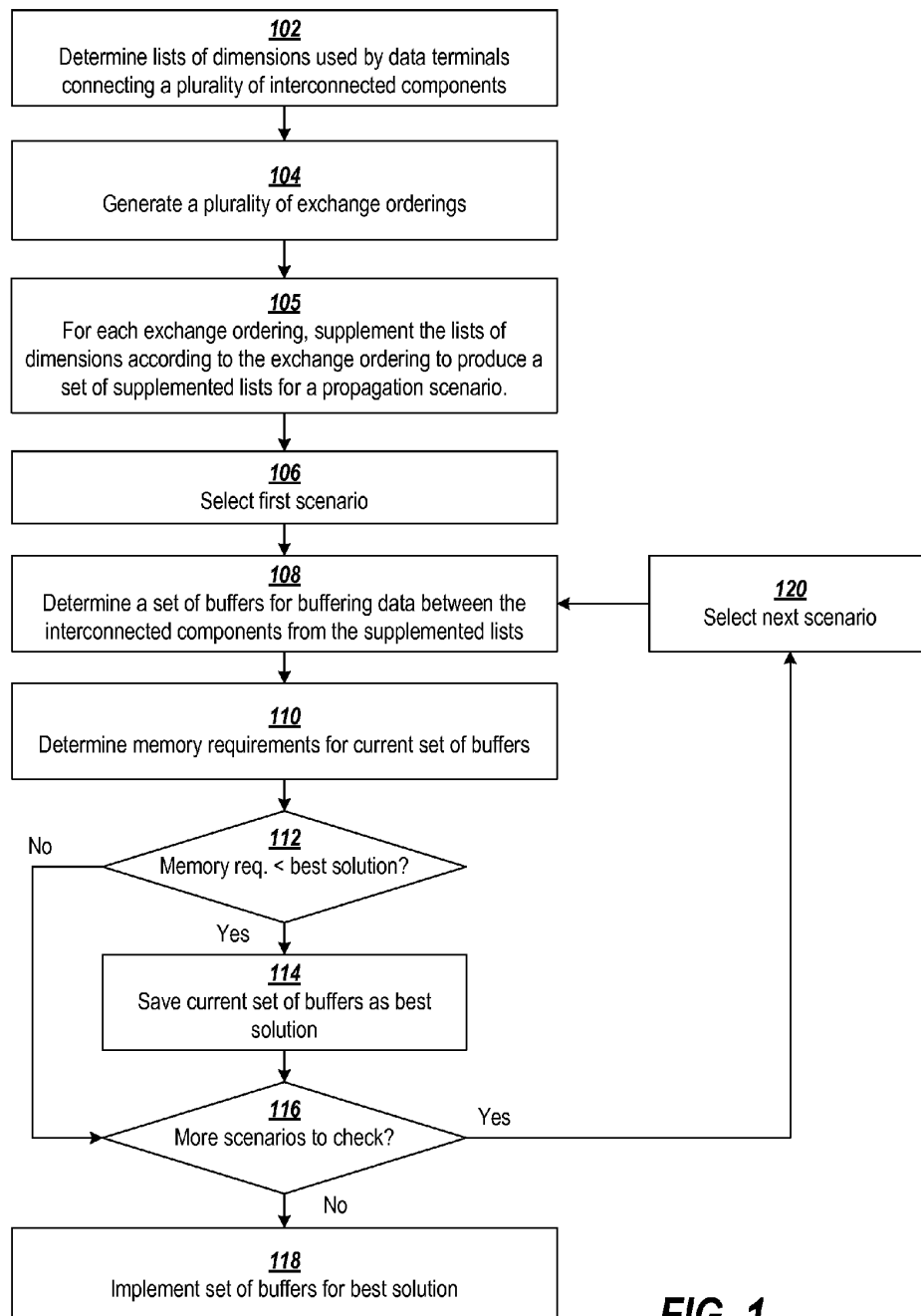
(74) *Attorney, Agent, or Firm* — LeRoy D. Maunu; Lois D.
Cartier

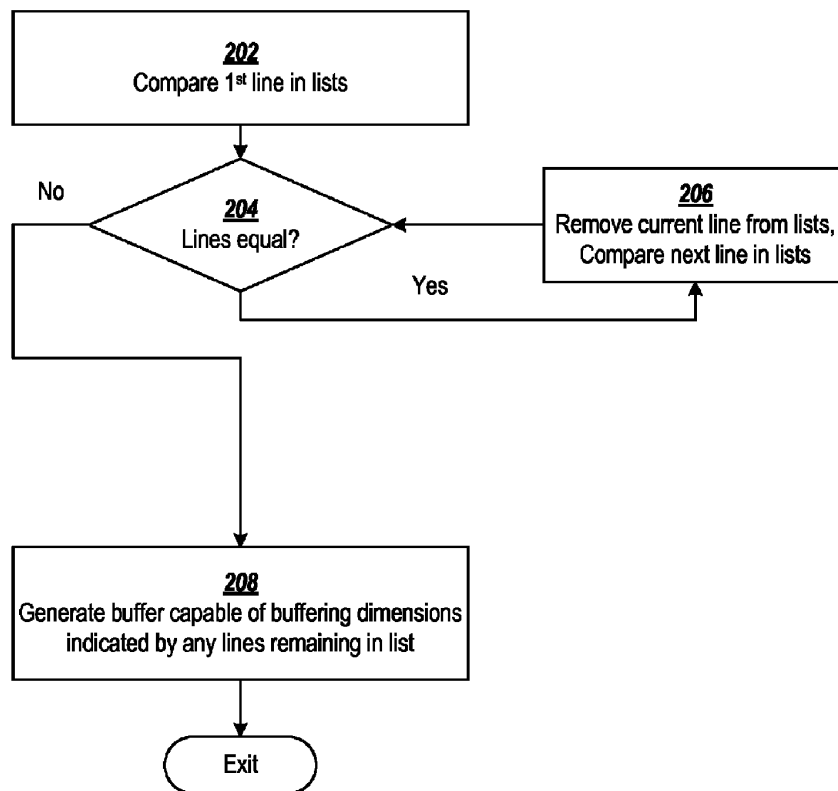
(57) **ABSTRACT**

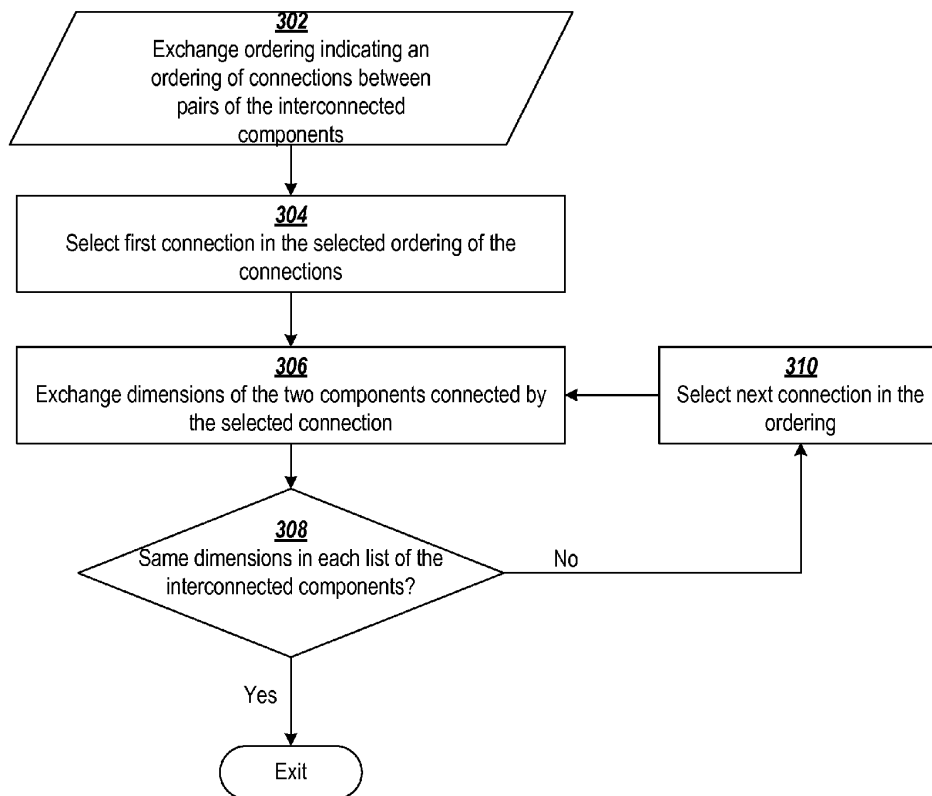
In one embodiment, a method of generating a circuit design is
provided. For each data terminal connecting a plurality of
components in a circuit design, a respective list of dimensions
of data used by the data terminal are determined. A plurality
of exchange orderings are generated that each indicate an
order in which dimensions are exchanged between the lists.
For each exchange ordering, dimensions are exchanged
between the lists according to the exchange ordering to pro-
duce a set of supplemented lists of dimensions. A set of
buffers for buffering data between the data terminals are
determined based on the supplemented lists of dimensions.
Memory requirements are determined for each of the set of
buffers. The circuit design is modified to include the one of
the determined sets of buffers having a lowest memory
requirement.

17 Claims, 14 Drawing Sheets



**FIG. 1**

**FIG. 2**

**FIG. 3**

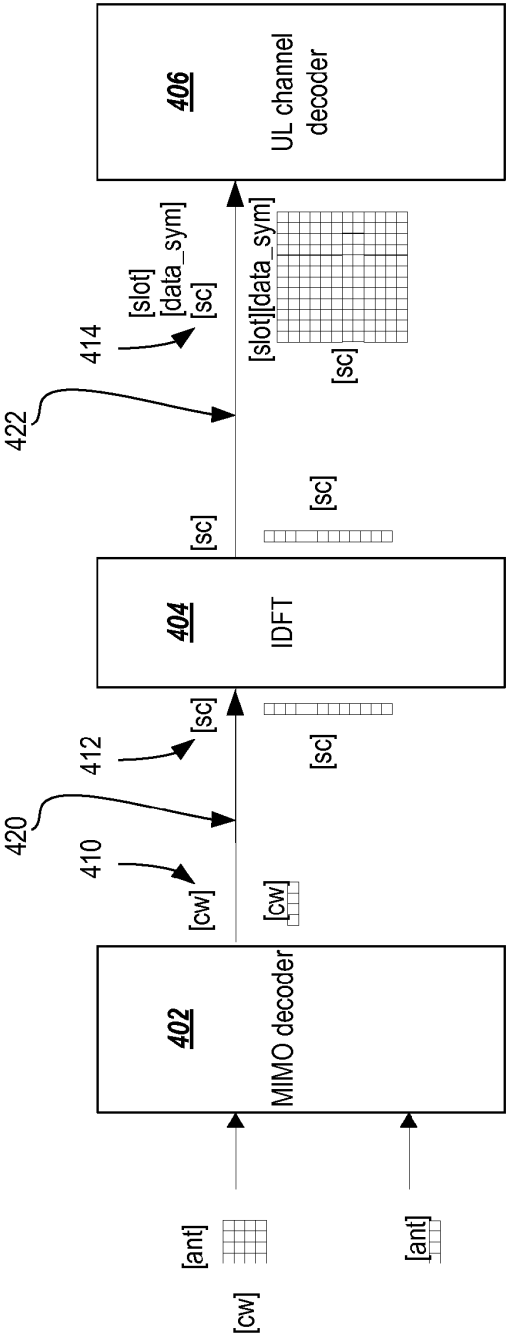


FIG. 4

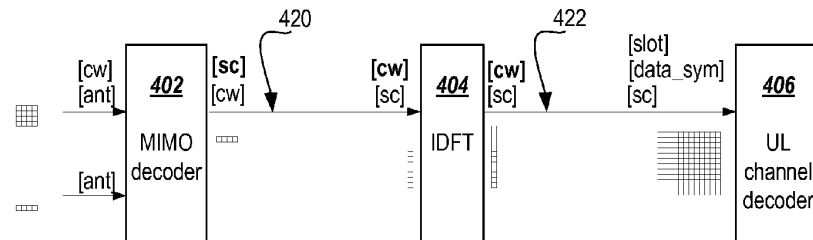
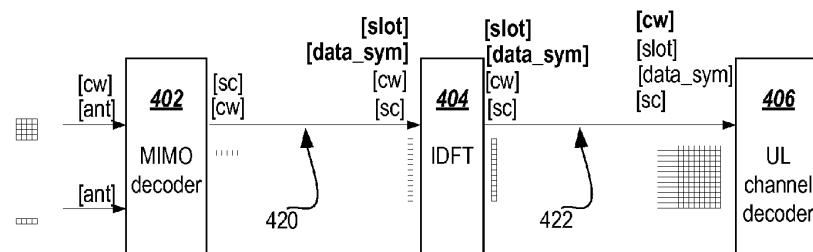
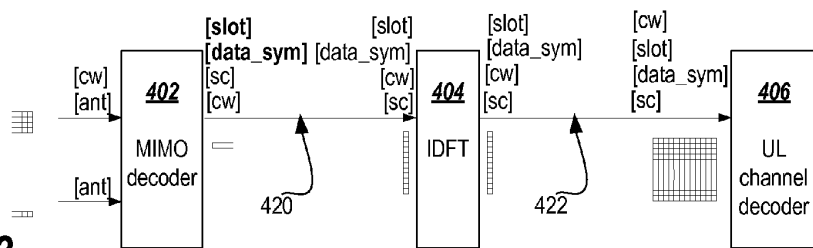
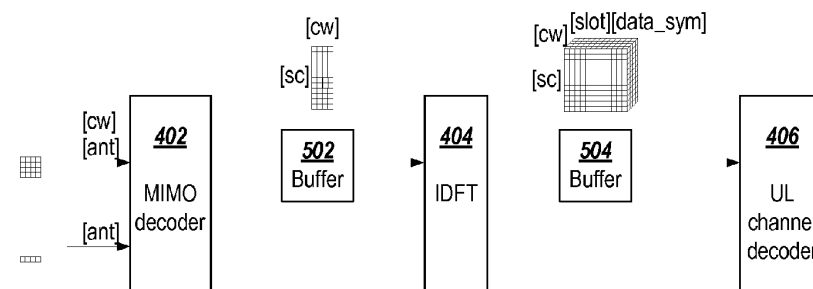
**FIG. 5-1****FIG. 5-2****FIG. 5-3****FIG. 5-4**

FIG. 6-1

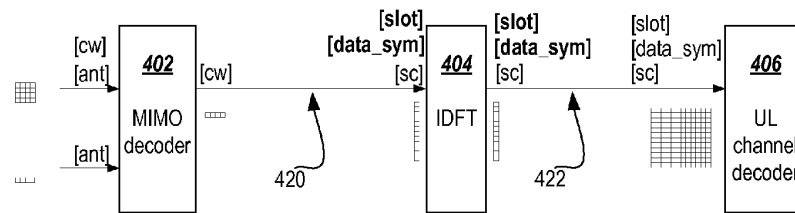


FIG. 6-2

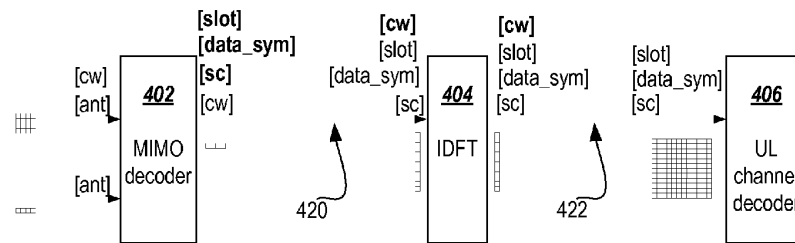


FIG. 6-3

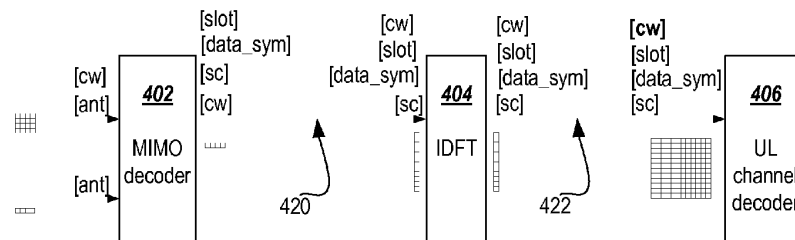
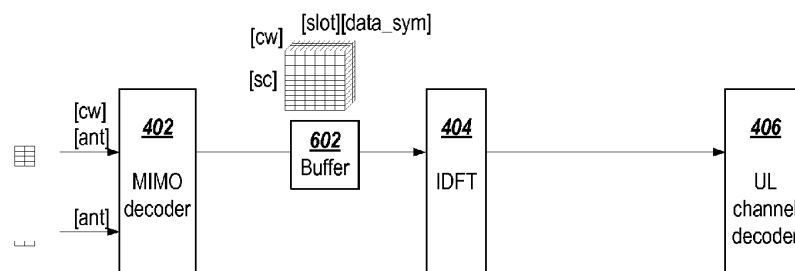
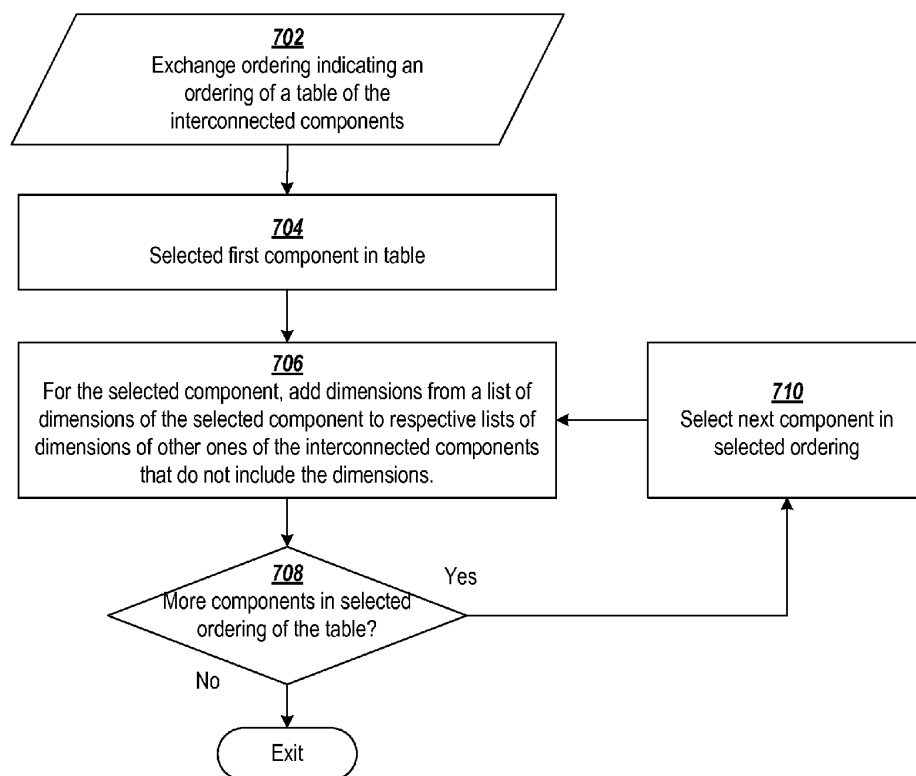
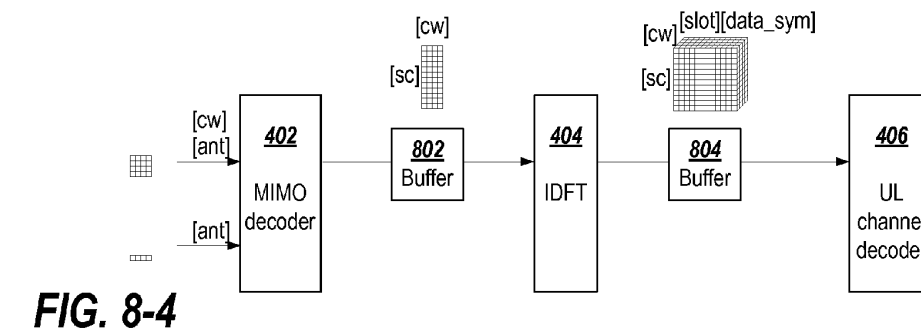
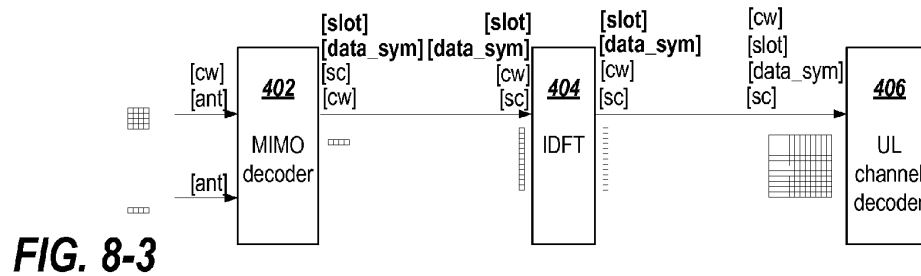
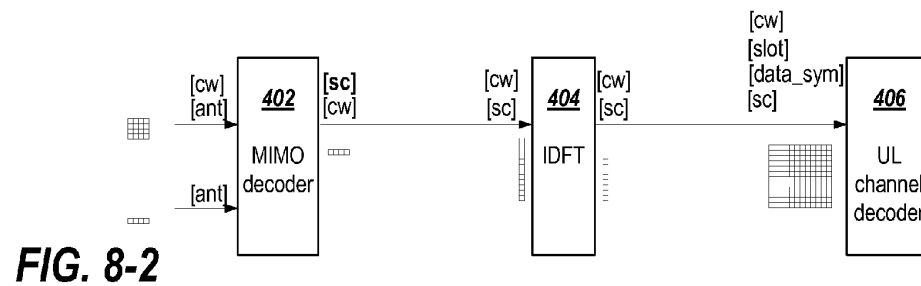
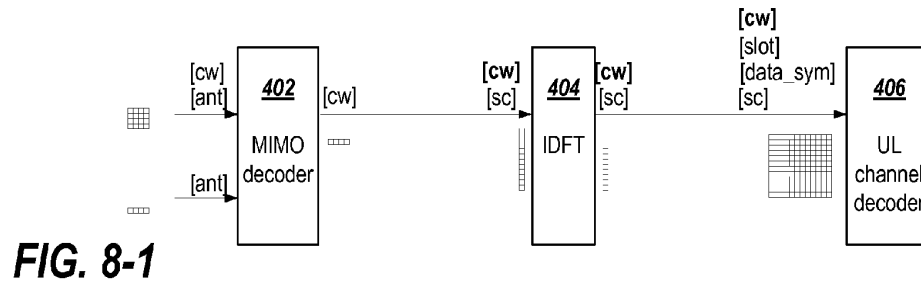
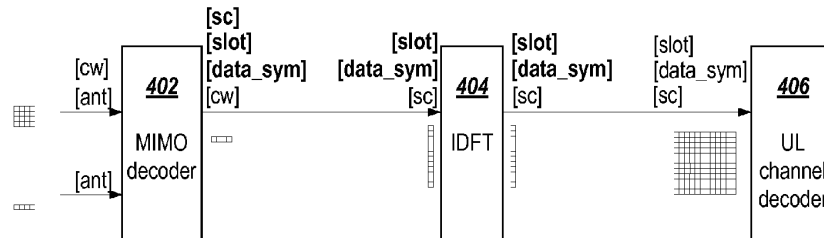
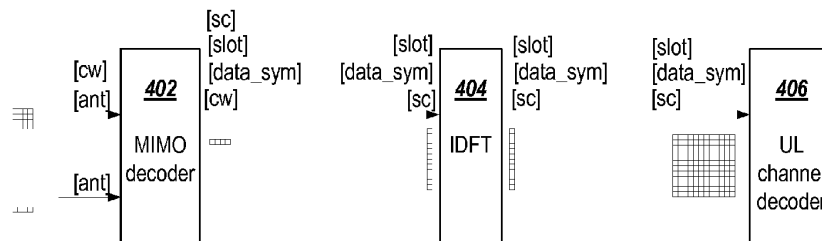
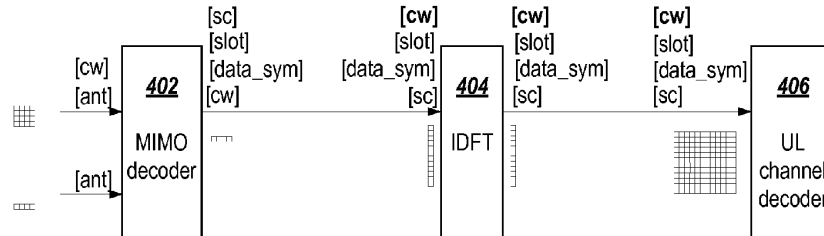
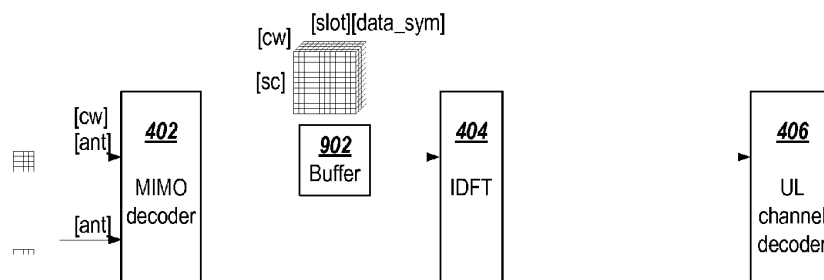


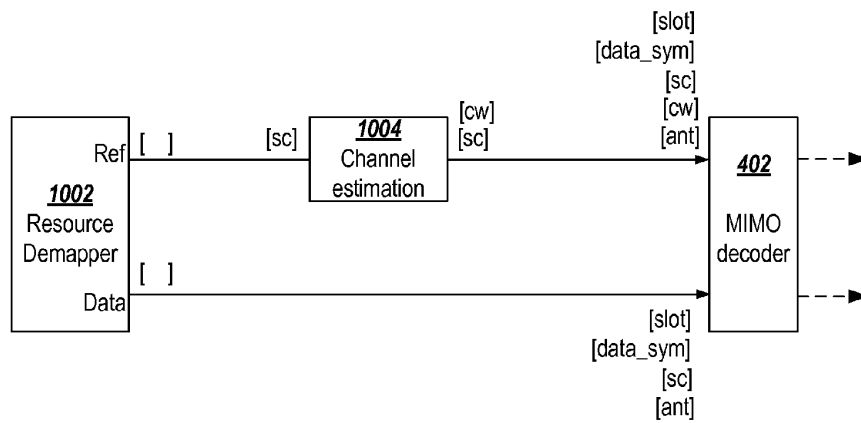
FIG. 6-4



**FIG. 7**



**FIG. 9-1****FIG. 9-2****FIG. 9-3****FIG. 9-4**

**FIG. 10**

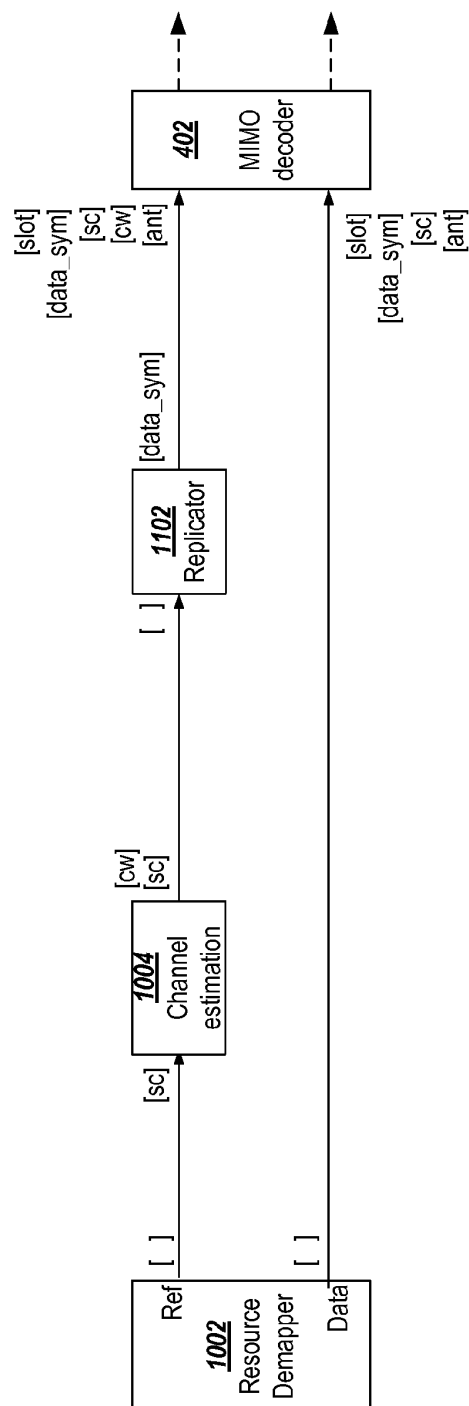


FIG. 11

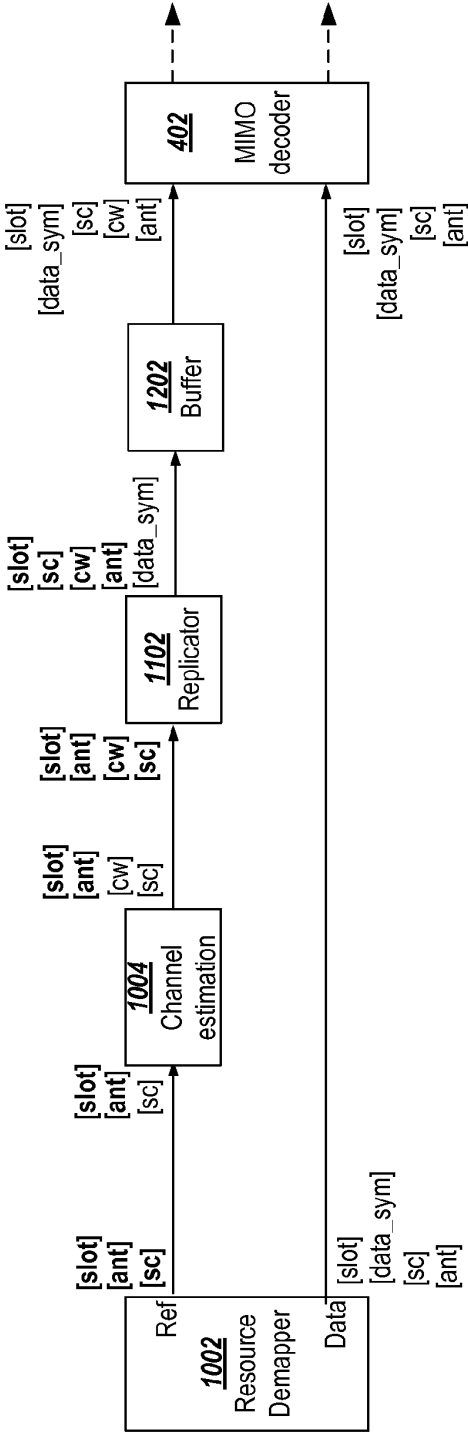


FIG. 12

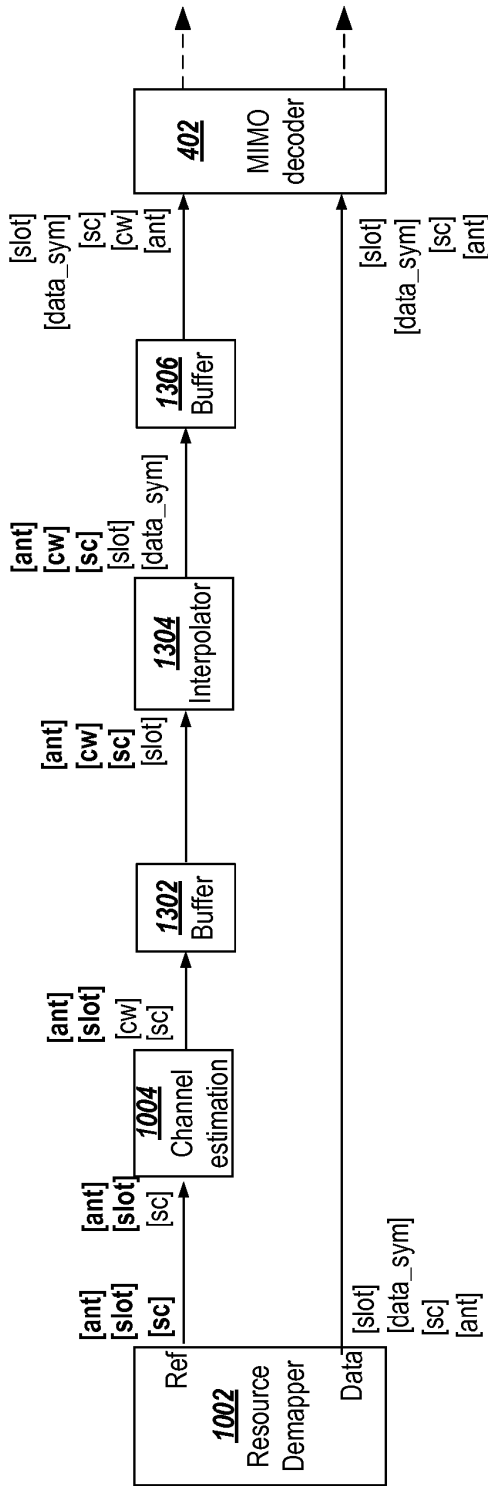
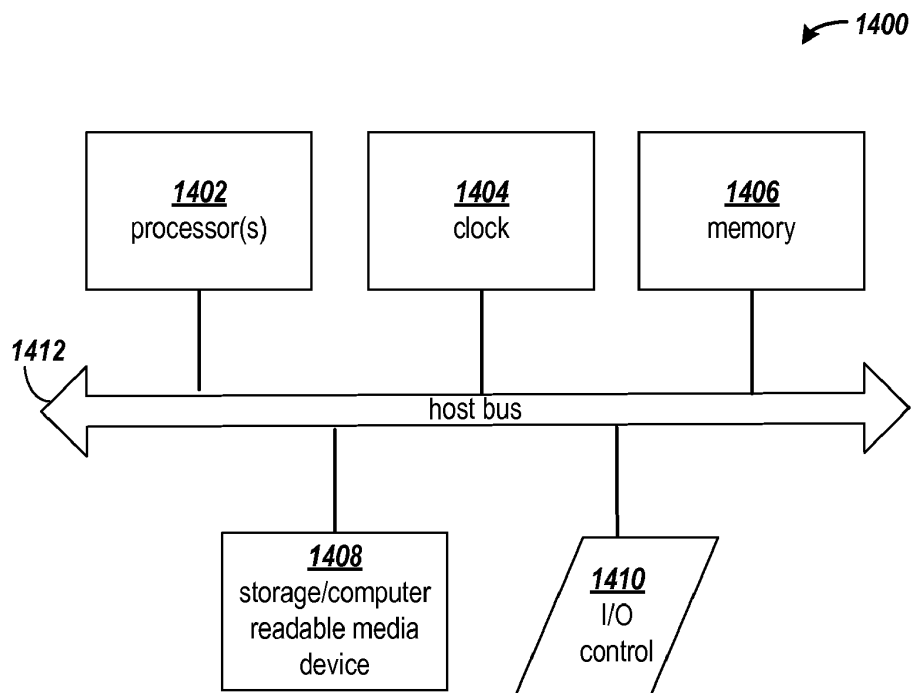


FIG. 13

**FIG. 14**

1

DETERMINING EFFICIENT BUFFERING FOR MULTI-DIMENSIONAL DATASTREAM APPLICATIONS

FIELD OF THE INVENTION

One or more embodiments generally relate to inter-circuit communication.

BACKGROUND

Complex integrated circuits (ICs) can be designed using various levels of abstraction. Using a hardware description language (HDL), circuits can be designed at the gate level, the register transfer level (RTL), and higher logical levels. When designing using an HDL, the designer describes the behavior of a system in terms of signals that are generated and propagated from one set of registers to another set of registers through combinatorial logic modules. HDLs provide a rich set of constructs to describe the functionality of each module. Modules may be combined and augmented to form even higher-level modules.

System-level integration may rely on reuse of previously created designs that have been provided either from within an enterprise or from a commercial provider. Libraries of pre-developed blocks of logic have been developed that can be selected and included in a circuit design. Such logic blocks include, for example, adders, multipliers, filters, and other arithmetic and digital signal processing (DSP) functions from which system designs can be readily constructed. The logic blocks may further include memories and storage elements. The engineering community sometimes refers to these previously created logic blocks as “design modules,” “cores,” “IP cores” (intellectual property cores), or “logic cores,” and such terms may be used interchangeably herein. The use of pre-developed IP cores permits faster design cycles by eliminating the redesign of circuits. Thus, using IP cores from a library may reduce design costs. Such IP cores may often be available for purchase by parties who desire the functionality provided by the IP core. IP cores include a circuit design in the form of source code or a netlist that may be used in implementing the design in a programmable IC, such as a field programmable gate array (FPGA). The core may be integrated into a design by instantiating the code or netlist. The logic core is then placed and routed along with the rest of the design to provide the desired functionality.

Incorporation of a logic core into a larger design, however, may not be a simple task. For example, different IP cores included in a signal processing design may be configured to operate on different dimensions of a data stream. For instance, in some communication systems, data may be received as codewords over a number of antennas, over a number of channel frequency bands, and/or over a number of time slots, etc. A first component of the signal processing design may be configured to output data as a set of codewords, whereas a next component of the signal processing design may be configured to receive data in the frequency domain as a set of subcarriers.

Due to the mismatch between the components, the components may not be able to be connected together directly without reformatting the data. To accommodate the mismatch in formats between the two components, sets of codewords may be buffered until complete sets of sub-carriers are obtained for input to the second component. Manual implementation of such buffers to perform such reformatting can be an error prone and time-consuming process. Further, such manual implementation requires designers to possess a requisite

2

knowledge of data format requirements of each of the components in the signal processing design, which may be cumbersome to obtain.

SUMMARY

In one embodiment, a method of generating a circuit design is provided. A circuit design having a plurality of components is received. Each component in the plurality of components has a data terminal connected to a data terminal of another one of the components. For each of the data terminals, a respective list of dimensions of data used by the data terminal are determined. Using at least one programmed processor, a plurality of exchange orderings are generated. Each exchange ordering indicates an order in which dimensions are exchanged between the lists. For each exchange ordering, dimensions are exchanged between the lists in the order indicated by the exchange ordering to produce a set of supplemented lists of dimensions. A set of buffers for buffering data between connected ones of the data terminals are determined based on the supplemented lists of dimensions. Memory requirements are determined for each of the set of buffers corresponding to each exchange ordering. Using the at least one programmed processor, the circuit design is modified to include the one of the determined sets of buffers having a lowest memory requirement.

In another embodiment, a method is provided. A specification having a plurality of components is received. Each of the plurality of components has a data terminal connected to a data terminal of another one of the components. For each of the data terminals, a respective list of dimensions of data used by the data terminal is determined. Using at least one programmed processor, a plurality of propagation scenarios are generated. Each propagation scenario indicates a set of supplemented lists resulting from a respective exchange of dimensions between the lists of dimensions such that each supplemented list in the set includes every dimension included in the lists of dimensions. The respective exchange of dimensions for each propagation scenario has a different order than the exchanges of scenarios corresponding to other ones of the plurality of propagation scenarios. For each propagation scenario, a respective set of buffers for buffering data between connected ones of the data terminals are determined based on the corresponding supplemented list of dimensions. Memory requirements are determined for each of the set of buffers corresponding to each exchange ordering. Using the at least one programmed processor, the specification is modified to include the one of the determined sets of buffers having a lowest memory requirement.

In yet another embodiment, a system is provided. The system includes a plurality of components, each having a respective list of dimensions indicating dimensions of a data stream utilized by the component. The plurality includes a first component that having a first data terminal and a second component that having a second data terminal. The system also includes at least one buffer coupled to the first data terminal and the second data terminals and configured to buffer the data stream transmitted between the first and second components. The at least one buffer is characterized by a memory requirement that is determined based on supplemental lists of dimensions that are generated by exchanging dimensions between the plurality of the lists of dimensions to supplement the first and second lists of dimensions.

It will be appreciated that various other embodiments are set forth in the Detailed Description and Claims, which follow.

BRIEF DESCRIPTION OF THE DRAWINGS

Various aspects and advantages of one or more embodiments will become apparent upon review of the following detailed description and upon reference to the drawings in which:

FIG. 1 shows a flowchart of an automated process for generating a set of buffers for data format conversion, in accordance with one or more embodiments;

FIG. 2 shows a flowchart of a process for determining required dimensions for buffering data between two components, in accordance with one or more embodiments;

FIG. 3 shows a flowchart of a process for exchanging dimensional data to produce supplemental lists of dimensions used in the process shown in FIG. 2;

FIG. 4 shows a block diagram of part of the physical layer of an LTE system that requires data format to be converted between the interconnected components;

FIGS. 5-1 through 5-4 illustrate exchange of dimensions between components in a first order using the process shown in FIG. 3 to generate a first buffer solution;

FIGS. 6-1 through 6-4 illustrate exchange of dimensions between components in a second order using the process shown in FIG. 3 to generate a second buffer solution;

FIG. 7 shows a flowchart of another process for exchanging dimensional data to produce supplemental lists of dimensions used in the process shown in FIG. 2;

FIGS. 8-1 through 8-4 illustrate exchange of dimensions between components in a first order using the process shown in FIG. 7 to generate a first buffer solution;

FIGS. 9-1 through 9-4 illustrate exchange of dimensions between components in a second order using the process shown in FIG. 7 to generate a second buffer solution;

FIG. 10 shows an example circuit that may precede the circuit shown in FIG. 4 in an LTE system;

FIGS. 11 and 12 illustrate an additional modification to the circuit shown in FIG. 10 that may be performed by a design tool to avoid unnecessary processing;

FIG. 13 illustrates another modification to the circuit shown in FIG. 10 that may be performed by a design tool to avoid unnecessary processing; and

FIG. 14 illustrates a block diagram of a computing arrangement on which several processes described herein may be implemented.

DETAILED DESCRIPTION

Connecting various components in a circuit design can be a complex task. For example, different IP cores included in a signal processing design may be configured to process different dimensions of a data stream. To accommodate the mismatch of dimensions of data utilized by different components, a data stream output from a first component may be required to be buffered to make a required dimension of a data stream available for further processing by a second component. Manual implementation of such buffers to perform such reformatting can be error prone and time-consuming. For instance, manual implementation of buffers for reformatting a data stream requires knowledge of data format requirements of each of the components in the signal processing design. However, such documentation may not be readily available or include the necessary format requirements. Furthermore, utilized dimensions of a data stream may not be defined for some customized components in a circuit design. In one or more embodiments, type information associated with component interfaces is collectively used to infer dimensions of data and desirable buffer sizes.

In one or more embodiments, buffer circuits are automatically generated and added to a circuit design based on dimensions of a data stream used by each of the interconnected components. For each data terminal that connects two interconnected components, a list of required dimensions is determined. Dimensions are exchanged between the lists of dimensions of the interconnected components to produce supplemented lists. The supplemented lists each include all dimensions used by the interconnected components. As discussed in more detail below, dimensions of data needed to be buffered between connected components can be determined based on the order in which dimensions appear in the lists of dimensions.

The order in which dimensions appear in the supplemented lists may be affected by the order in which dimensions are exchanged. As such, depending on the ordering in which dimensions are exchanged between the components, different sets of dimensions for the buffers may be identified. Accordingly, several different sets of buffers may each be sufficient to reformat a data stream for a set of components operating on different dimensions of a data stream. However, the different sets of buffers may have different memory resource requirements for implementation. The number of these different possible sets of buffer arrangements may prohibit manual analysis of memory requirements of the different solutions. In one or more embodiments, a plurality of different buffer arrangements are generated and analyzed to determine memory resource requirements of each solution. For ease of reference, each of the different sets of buffers may be referred to as a buffer solution. Code is generated and added to the design to implement the buffer solution requiring the least amount of memory resources.

FIG. 1 shows a flowchart of an example method for determining memory-efficient buffers for reformatting a data stream between components operating on different dimensions of a data stream. The process determines a plurality of buffer solutions. Based on memory requirements of the plurality of buffer solutions, a set of memory efficient buffers is determined.

For each of a plurality of interconnected components, a list of dimensions of a data stream used by data terminals connecting the interconnected components is determined at block 102. As indicated above, different buffer solutions may be generated by exchanging dimensions between components in different respective orders. For ease of reference, each different ordering in which dimensions can be exchanged between lists may be referred to as an exchange ordering. When dimensions are exchanged between lists according to an exchange ordering, each list of dimensions is fully supplemented to include each dimension of a data stream used by the plurality of interconnected components. For ease of reference, the fully supplemented lists may be referred to as a propagation scenario and such terms are used interchangeably herein.

A plurality of exchange orderings are generated at block 104. For each exchange ordering, dimensions are exchanged between the lists of dimensions according to the exchange ordering at block 105 to produce supplemented lists for a respective propagation scenario. As explained in more detail with respect to FIGS. 3 and 7, dimensions may be exchanged between the lists of dimensions using a number of different processes.

A first one of the propagation scenarios is selected at block 106. Using the supplemented lists of dimensions in the selected propagation scenario, a set of buffers for buffering data between the interconnected components is determined at block 108. Memory resource requirements of the set of buff-

5

ers are determined at block 110. If the determined memory resource requirements are less than the best solution at decision block 112, the current set of buffers is saved as the best solution at block 114. While there are more propagation scenarios to check as determined at decision block 116, the next scenario is selected at block 120, and a buffer solution is determined as described above. After all scenarios have been checked at decision block 116, code may be generated at block 118 to implement the set of buffers having the least memory requirements (i.e., the set of buffers saved as the best solution).

FIG. 2 shows a flowchart of an example process for determining required dimensions for buffering between two components based on the supplemented lists of dimensions of a propagation scenario. In one or more embodiments, the process may be used to determine a set of buffers in block 108 of FIG. 1. As indicated above, the order in which dimensions appear in the supplemented lists of two connected components (for a propagation scenario) indicates which dimensions of the data stream are required to be buffered. The process shown in FIG. 2 assumes that a list of dimensions are supplemented by adding new dimensions to the top of the list. Under such constraints, a dimension located at the top of both lists is not required to be buffered.

The first line in a first supplemented list of a propagation scenario is compared to a first line of the second list of the propagation scenario at block 202. If the compared lines indicate the same dimension as determined at decision block 204, the line is removed from each of the lists and the next line in the two lists are compared at block 206. If the compared lines indicated different dimensions as determined at decision block 204, each remaining dimension indicated by the lists must be buffered. At block 208, a buffer is generated that is capable of buffering dimensions of the data stream indicated by lines remaining in the lists.

Although the process shown in FIG. 2 assumes that the lists of dimensions are supplemented by adding new dimensions to the top of the list, the embodiments are not so limited. For instance, it is recognized that the processes may be configured to process the lists of dimensions having dimensions arranged in reverse format (e.g. supplemented by adding dimensions to the bottom of the list).

FIG. 3 shows a flowchart of a process for exchanging dimensional data between lists of dimensions to produce supplemental lists of dimensions for one propagation scenario. In this example process, an input exchange ordering 302 indicates an ordering of connections between multiple interconnected components. For ease of reference an ordering of connections may be referred to as a connection ordering. A first one of the connections indicated by the ordering of the scenario 302 is selected at block 304. Dimensions are exchanged between dimension lists of two components connected by the connection at block 306. The exchange of dimensions may be performed by supplementing each respective list with dimensions included in the other. For instance, if a dimension is indicated by a first one of the two lists but not the second list, the dimension is added to the second list. Dimensions from a first list are added to another list in the same order that the dimensions appear in the first list. The next connection in the connection ordering is selected at block 310 and dimension exchange is repeated at block 306 until the same set of dimensions are indicated in each of the supplemented lists at decision block 308.

The exchange of dimensions between the lists of dimensions and the generation of different buffer solutions may be explained by way of example. FIG. 4 shows a block diagram of part of the physical layer of an LTE system having a series

6

of interconnected components that operate on different dimensions of a data stream. The interconnected components include a MIMO decoder 402, an inverse discrete Fourier transform (IDFT) 404, and a UL channel decoder 406. In the LTE system, the MIMO decoder 402 identifies senders (associated with different codewords) using samples received from multiple receive antennas. The MIMO decoder 402 is able to process one resource element at a time, where a resource element is a data sample at a particular frequency and time. For each resource element, it takes a small channel matrix having codeword [cw] and antenna [ant] dimensions, as well as a sample of data across the antennas [ant] to generate an output for a number of codewords [cw]. This data is then processed by the IDFT 404. However, the IDFT is applied across a group of subcarriers [sc] for a single codeword. Due to the mismatch between the dimensions of a data stream operated on by the components 402 and 404, a buffer is required to allow the data stream to be reformatted. Similar mismatch between dimensions of the data stream occurs between the IDFT component 404 and the UL channel decoder 406, which is connected to the output of the IDFT 404. In this example, the UL channel decoder 406 operates on a data block including a number of time domain slots [slot], which each include a number of data symbols [data_sym], across a number of subcarriers [sc]. Each of the components 402, 404, and 406 has a respective list of dimensions 410, 412, and 414 used by data terminals that interconnect the components. For ease of illustration and explanation regarding exchange of dimensions between lists, the list of dimensions 414 for component 404 is illustrated separately for each of the two connections 420 and 422.

FIGS. 5-1 through 5-4 illustrate exchange of dimensions between lists of dimensions for components shown in FIG. 4, and the exchange of dimensions is in a first connection order using the process shown in FIG. 3 to generate a first buffer solution. In this example, dimensions are exchanged across connections in the order of {420, 422}, which is repeated until each of the lists of dimensions contain the same dimensions. FIGS. 5-1 illustrates an exchange of dimensions between lists of dimensions of components 402 and 404 for a first selected connection 420 of the ordering {420, 422}. As a result of the exchange, the list of component 402 is supplemented with dimension [sc] and the list of component 404 is supplemented with dimension [cw] (supplemented dimensions are indicated in bold). For ease of explanation and illustration, supplementation of dimensions of inputs to MIMO decoder 402 are not shown in FIGS. 5-1 through 5-4. After the exchange in FIGS. 5-1, the three lists 410, 412, and 414 include different dimensions. As such, exchanging of dimensions continues as described in connection with FIG. 3.

FIGS. 5-2 illustrates an exchange of dimensions between lists of dimensions of components 404 and 406 for a second selected connection 422 of the ordering {420, 422}. As a result of the exchange, the list of component 404 is supplemented with dimensions [slot] and [data_sym] and the list of component 406 is supplemented with dimension [cw]. After the exchange in FIGS. 5-2, the three lists 410, 412, and 414 still include different dimensions, so exchanging of dimensions continues with the connection ordering {420, 422} restarted.

FIGS. 5-3 illustrates an exchange of dimensions between lists of dimensions of components 402 and 404 for the first selected connection 420 of the ordering {420, 422}. As a result of the exchange, the list of component 402 is supplemented with dimensions [slot] and [data_sym] and all of the supplemented lists include the same dimensions. Therefore,

as indicated in FIG. 3, exchanging of dimensions for the connection ordering {420, 422} is completed.

FIGS. 5-4 illustrates generation of a set of buffers (i.e. a buffer solution) for the propagation scenario resulting from connection ordering {420, 422} based on the supplemented lists of dimensions shown in FIGS. 5-3. As described with reference to FIG. 2, matching dimensions located at the top of both lists are not required to be buffered. For connection 420, the lists of connected components 402 and 404 both include dimensions [slot] and [data_sym] in the same location at the top of the list. As such, only dimensions [sc] and [cw] are required to be buffered in buffer 502. For connection 422, each of the dimensions must be buffered in buffer 504 because the first line in the lists of connected components 404 and 406 do not match.

As briefly indicated above, the buffer solutions generated may depend on the order in which dimensions are exchanged between lists for the different scenarios. FIGS. 6-1 through 6-4 illustrate exchange of dimensions between lists of dimensions, for components shown in FIG. 4, in a second order using the process shown in FIG. 3 to generate a second buffer solution. In this example, dimensions are exchanged across connections in the connection ordering {422, 420}, which is repeated until each of the lists of dimensions contain the same dimensions. FIGS. 6-1 illustrates an exchange of dimensions between lists of dimensions of components 404 and 406 for the first connection 422 of the ordering {422, 420}. As a result of the exchange, the list of dimensions for component 404 is supplemented with dimensions [slot] and [data_sym]. For ease of explanation and illustration, supplementation of dimensions of inputs to MIMO decoder 402 are not shown in FIGS. 6-1 through 6-4. After the exchange in FIGS. 6-1, the three lists 410, 412, and 414 include different dimensions. As such, exchanging of dimensions continues as described in connection with FIG. 3.

FIGS. 6-2 illustrates an exchange of dimensions between lists of dimensions of components 402 and 404 for a second selected connection 420 of the connection ordering {422, 420}. As a result of the exchange, the list of dimensions for component 402 is supplemented with dimensions [slot], [data_sym], and [sc] and the list of component 404 is supplemented with dimension [cw]. After the exchange in FIGS. 6-2, the three lists 410, 412, and 414 still include different dimensions, so exchanging of dimensions continues with the connection ordering {422, 420} restarted.

FIGS. 6-3 illustrates an exchange of dimensions between lists of dimensions of components 404 and 406 for the first selected connection 422 of the connection ordering {422, 420}. As a result of the exchange, the list of dimensions for component 406 is supplemented with dimensions [cw] and all of the supplemented lists include the same dimensions. Therefore, as indicated in FIG. 3, exchanging of dimensions for the connection ordering {420, 422} is completed.

FIGS. 6-4 illustrates generation of a set of buffers (i.e. a buffer solution) for the scenario order resulting from the connection ordering {422, 420} based on the supplemented lists of dimensions shown in FIGS. 6-3. As described with reference to FIG. 2, dimensions located at the top of both lists are not required to be buffered. For connection 420, each of the dimensions must be buffered in buffer 602 because the first line in the lists of connected components 402 and 404 do not match. For connection 422, none of the dimensions need to be buffered because each of the lines in the lists of components 404 and 406 match.

In comparison between the buffer solutions shown in FIGS. 5-4 and FIGS. 6-4, it can be seen that buffer 504 requires the same amount of memory resources as buffer 602.

However, the buffer solution of FIGS. 5-4 also requires memory resources to implement buffer 502. Therefore, the buffer solution generated in FIGS. 6-2 as a result of the connection ordering {422, 420} is the more optimal buffer solution.

FIG. 7 shows a flowchart of another process for exchanging dimensions to produce the supplemental lists of dimensions for a particular order indicated by an exchange ordering. In this example process, an input exchange ordering 702 indicates an order of a table of the interconnected components. For ease of reference, an ordering of the interconnected components may be referred to as a component ordering. A first one of the components indicated by the component ordering 702 is selected at block 704. Dimensions from a list of dimensions of the selected component are used to supplement lists of dimensions for other components at block 706. For instance, a dimension in a selected list is added to one of the other list of dimensions if the dimension is not included in the other list of dimensions. Dimensions added from a first list to a second list are added in the same order that the dimensions appear in the first list. The next component in the component ordering 702 is selected at block 710 and dimension exchange is repeated at block 706 until the same set of dimensions are indicated in each of the supplemented lists at decision block 708.

FIGS. 8-1 through 8-4 illustrate exchange of dimensions between the example components shown in FIG. 4 in a first component ordering using the process shown in FIG. 7 to generate a first buffer solution. In this example, dimensions are supplemented from lists of dimensions in the component ordering {402, 404, and 406}. This process only requires that dimensions be supplemented from each list once to produce the fully supplemented lists of a propagation scenario. For ease of explanation and illustration, supplementation of dimensions of inputs to MIMO decoder 402 are not shown in FIGS. 8-1 through 8-4.

FIGS. 8-1 illustrates supplementing dimensions from the list of dimensions for the first component 402 in the component ordering {402, 404, 406} to the lists of components 404 and 406. As a result, the lists of dimensions for components 404 and 406 are each supplemented with dimension [cw]. FIGS. 8-2 illustrates the supplementation of dimensions from the list of dimensions for the second component 404 in the ordering {402, 404, and 406}. As a result, the list of dimensions for component 402 is supplemented with dimension [sc]. FIGS. 8-3 illustrates the supplementation of dimensions from the list of dimensions for the last component 406 in the component ordering {402, 404, and 406}. As a result, the lists of components 402 and 404 are each supplemented with dimensions [slot] and [data_sym].

FIGS. 8-4 illustrates generation of a set of buffers (i.e. a buffer solution) for the propagation scenario resulting from the component ordering {402, 404, 406} shown in FIGS. 8-1 through 8-3 based on the supplemented lists of dimensions shown in FIG. 8-3. As described with reference to FIG. 2, dimensions located at the top of both lists are not required to be buffered. Similar to the solution shown in FIGS. 5-4, the lists of connected components 402 and 404 both include dimensions [slot] and [data_sym] in the same location at the top of the list. As such, only dimensions [sc] and [cw] are required to be buffered in buffer 802. For connection 422, each of the dimensions must be buffered in buffer 804 because the first line in the lists of connected components 404 and 406 do not match.

As described above, the buffer solutions generated may depend on the component ordering in which dimensions are supplemented from selected components to other compo-

nents for the different scenarios. FIGS. 9-1 and 9-2 illustrate exchange of dimensions between components shown in FIG. 4 in a second order using the process shown in FIG. 7 to generate a second buffer solution. In this example, dimensions are supplemented from lists of dimensions in the component ordering {406, 404, and 402}. For ease of explanation and illustration, supplementation of dimensions of inputs to MIMO decoder 402 are not shown in FIGS. 9-1 through 9-4.

FIGS. 9-1 illustrates supplementing dimensions from the list of dimensions for the first component 406 in the component ordering {406, 404, 402} to the lists of dimensions for components 402 and 404. As a result, the lists of dimensions for components 402 and 404 are each supplemented with dimensions [sc], [slot], and [data_sym]. FIGS. 9-2 illustrates the supplementation of dimensions from the list of dimensions for the second component 404 in the component ordering {406, 404, and 402}. In this instance, the list of dimensions for component 404 does not include any dimensions that are not included in lists of dimensions for components 402 and 406. Therefore, the lists of dimensions are not supplemented from component 404. FIG. 9-3 illustrates the supplementation of dimensions from the list of the last component 402 in the component ordering {406, 404, and 402}. As a result, the lists of dimensions for components 404 and 406 are each supplemented with the dimension [cw].

FIGS. 9-4 illustrates generation of a set of buffers for the propagation scenario resulting from component ordering {406, 404, and 402} based on the supplemented lists of dimensions shown in FIGS. 9-3. As described above, dimensions located at the top of both lists are not required to be buffered. Similar to the solution shown in FIGS. 6-4, for connection 420, each of the dimensions must be buffered in buffer 902 because the first line in the lists of connected components 402 and 404 do not match. For connection 422, none of the dimensions need to be buffered because each of the lines in the lists of components 404 and 406 match.

While the exchange of dimensions between lists of dimensions is primarily described with reference to the processes shown in FIGS. 3 and 7, it is recognized that other processes for exchanging dimensions to produce the supplemented lists of dimensions may be used as well.

As indicated above, manual implementation of buffers for reformatting a data stream requires knowledge of data format requirements of each of the components in the signal processing design. However, documentation for IP core implemented sub-circuits may not be readily available or include the necessary format requirements. In one or more embodiments, a computer-assisted tool is configured to automatically determine data format requirements of the different components of a signal processor and generate buffer circuitry for data conversion between the circuit components.

For instance, in some embodiments, data ports of an IP core may be determined from metadata stored in the IP core. Metadata is associated with each component describing the dimensionality of the data on its input and output interfaces. The meta-data is defined once, by the core developer, and is stored in a library of IP. By formally defining the interfaces on the components the time-consuming and error prone task of extracting this data from a textual or graphical description in the datasheet is avoided. For example, using data port definitions retrieved from the meta-data and using a mapping of the circuit design between the data port and variables of the circuit design, dimensions of a data stream used by the IP core can be determined. Alternatively, in some embodiments, the data ports of an IP-core may be retrieved from an offline or online database provided by a distributor of the IP-core.

It is understood that input and output data formats may not be defined for all components of a circuit design. For example, a designer may include one or more custom designed components that interoperate with components implemented with IP cores. In one or more embodiments, the design tool may be configured and arranged to assist in the definition of a custom component based on dimensions of a data stream used by components for which the utilized dimensions have been determined. In one or more embodiments, suggested dimensions for a custom component may be determined from an identified buffer solution for components implemented with IP-cores. For instance, dimensions of a customized component and components connected thereto can be exchanged as described above to determine dimensions required by the adjacent components for the buffer solution. Based on such determination of dimensions, the design tool can automatically determine a set of recommended dimensions for the custom component that will require a least amount of additional buffering. The recommended dimensions may be displayed to a designer to assist in defining the custom component. In some implementations, the design tool may provide an interface for the designer to review and modify the recommended dimensions to define a set of dimensions for the custom component. In some embodiments, the design tool may provide an interface for the designer to specify a minimum set of dimensions that are to be defined for the custom component. Such a minimum set of dimensions may be specified initially before buffer solutions are determined or after a recommended buffer solution is displayed to the designer. In some embodiments, in response to the designer indicating modifications of the recommended set of dimensions or specifying the minimum set of dimensions, the design tool is configured to recalculate a new best buffer solution that takes the designers modifications and/or indicated minimum set of dimensions into account.

An example of determining recommended dimensions for a custom component is illustrated with respect to FIG. 10. FIG. 10 shows an example circuit that may precede the circuit shown in FIG. 4 in an LTE system. Channel estimation component 1004 generates channel and noise estimates for the MIMO decoder component 402 shown in FIG. 4. Resource demapper 1002 is configured to read data from a multi-dimensional data block and output data from two interfaces: one for reference symbols (which are processed by the channel estimation component 1004 and one for data symbols (which are input to MIMO decoder 402). For ease of explanation, dimensions of the supplemented list of MIMO decoder 402 in FIGS. 9-2 are presumed to be a fixed constraint of the MIMO decoder 402 in FIG. 10. In this illustrative example, channel estimation component 1004 is implemented by an IP core defined to operate on the [sc] and [cw] dimensions and the resource demapper 1002 is a custom user designed component that does not have output data stream dimensions defined. Because the resource demapper 1002 is a custom component block, the dimensions on the output interfaces are not pre-defined. Rather, these are defined by a designer of the system and dictate how the resource demapper is thereafter implemented. The exchange of dimensions between the interconnected components, as described above, can be used to automatically define the data format of a custom component like resource demapper 1002. For instance, the data output of resource demapper 1002 can be defined to have the same data format as the connected input of the MIMO decoder 402.

However, if reference (ref) output of resource demapper 1002 is defined to use a data format having the list of the dimensions of the input of the MIMO decoder 402, the

11

resource demapper **1002** will be configured to use the [data_sym] dimension. This causes difficulty because the MIMO decoder **402** requires one estimate from the channel estimation per data symbol, and thus the channel estimation component **1004** must produce a channel estimate for each data symbol, and the resource demapper must produce a reference symbol for each data symbol. One way to handle this is to implement the resource demapper **1002** to replicate the reference symbol data. However, this approach requires channel estimates to be unnecessarily calculated repeatedly by the channel estimation component **1004** for the same reference symbol data.

FIG. **11** illustrates a modification to the circuit shown in FIG. **10** that avoids unnecessary calculation of the channel estimation component **1004**. The unnecessary calculation of the channel estimation component **1004** is avoided by providing a replicator block after the channel estimation component to provide the [data_sym] dimension used by the MIMO decoder **402**. The replicator block **1102** consumes a single element and produces copies of that element for each data symbol. However, because the replicator block and the other components operate on different dimensions of the data stream additional buffering is required. Suitable buffers may be generated using the processes described above. FIG. **12** shows the circuit of FIG. **11**, with buffers **1202** generated using supplemented lists of dimensions as described in the above processes. As in the above figures, dimensions added to each list of dimensions are indicated in bold. In this example, buffer **1202** is required to buffer dimensions [sc] [cw], [ant], and [data_sym] between replicator block **1102** and MIMO decoder **402**. In one or more embodiments, a design tool is configured to insert a replicator block to avoid replication of calculations in earlier components in the data flow. In some implementations insertion may be performed automatically by adding constraints to the dimensions of the ref output of the resource demapper **1002** that may be reordered. In another implementation, a design tool is configured to provide an interface that allows a designer to select a connection and insert a replicator circuit (e.g., **1102**).

FIG. **13** illustrates another modification to the circuit shown in FIG. **10** that may be performed by a design tool to replicate the reference symbol data while avoiding unnecessary calculation of the channel estimation component **1004**. In this example, channel estimates are interpolated between data slots of the data stream. To perform interpolation an interpolator block **1304** (having defined data type [slot]->[slot][data_sym]) is added between the channel estimation component **1004** and the MIMO decoder **402**. To match dimensions of the data stream used by the interpolator block **1304**, the channel estimation component **1004**, and the MIMO decoder **402**, lists of dimensions used by each component are generated, supplemented, and used to generate buffers **1302** and **1306** using, e.g., the process described in FIG. **1**.

FIG. **14** shows a block diagram of an example computing arrangement that may be configured to implement the processes and functions described herein. It will be appreciated that various alternative computing arrangements, including one or more processors and a memory arrangement configured with program code, would be suitable for hosting the processes and data structures and implementing the algorithms of the different embodiments. The computer code, comprising the processes of one or more embodiments encoded in a processor executable format, may be stored and provided via a variety of computer-readable storage media or

12

delivery channels such as magnetic or optical disks or tapes, electronic storage devices, or as application services over a network.

Although the embodiments are primarily described and illustrated with reference to buffering between components of a circuit design specification, the embodiments are not so limited. For instance, it is recognized that the embodiments could also be applied to buffering between different software components that are implemented in software. For such applications, the sets of buffers describe herein represents data conversions between the software modules. Because memory would be required to perform data conversions between the software modules, configuration of data conversions using one or more of the embodiments could assist to reduce the amount of memory required by the software application.

Processor computing arrangement **1400** includes one or more processors **1402**, a clock signal generator **1404**, a memory unit **1406**, a storage unit **1408**, and an input/output control unit **1410** coupled to a host bus **1412**. The arrangement **1400** may be implemented with separate components on a circuit board or may be implemented internally within an integrated circuit. When implemented internally within an integrated circuit, the processor computing arrangement is otherwise known as a microcontroller.

The architecture of the computing arrangement depends on implementation requirements as would be recognized by those skilled in the art. The processor **1402** may be one or more general-purpose processors, or a combination of one or more general-purpose processors and suitable co-processors, or one or more specialized processors (e.g., RISC, CISC, pipelined, etc.).

The memory arrangement **1406** typically includes multiple levels of cache memory, and a main memory. The storage arrangement **1408** may include local and/or remote persistent storage such as provided by magnetic disks (not shown), flash, EPROM, or other non-volatile data storage. The storage unit may be read or read/write capable. Further, the memory **1406** and storage **1408** may be combined in a single arrangement.

The processor arrangement **1402** executes the software in storage **1408** and/or memory **1406** arrangements, reads data from and stores data to the storage **1408** and/or memory **1406** arrangements, and communicates with external devices through the input/output control arrangement **1410**. These functions are synchronized by the clock signal generator **1404**. The resource of the computing arrangement may be managed by either an operating system (not shown), or a hardware control unit (not shown).

The disclosed embodiments are thought to be applicable to a variety of applications and tools related to circuit design and electronic circuitry. Other aspects and embodiments will be apparent to those skilled in the art from consideration of the specification. It is intended that the specification and illustrated embodiments be considered as examples only, with a true scope and spirit of the invention being indicated by the following claims.

What is claimed is:

1. A method, comprising:

receiving a circuit design having a plurality of components, each component having a data terminal connected to a data terminal of another one of the components; for each of the data terminals, determining a respective list of dimensions of data used by the data terminal; and performing on at least one programmed processor, operations including:

13

generating a plurality of exchange orderings, each exchange ordering indicating an order in which dimensions are exchanged between the lists of dimensions;

for each exchange ordering:

- exchanging dimensions between the lists in the order indicated by the exchange ordering to produce a set of supplemented lists of dimensions;
- determining a set of buffers for buffering data between connected ones of the data terminals based on the supplemented lists of dimensions; and
- determining memory requirements for the set of buffers corresponding to each exchange ordering; and
- modifying the circuit design to include the one of the determined sets of buffers having a lowest memory requirement.

2. The method of claim 1, wherein each of the plurality of exchange orderings indicates an order in which dimensions are exchanged between pairs of connected ones of the plurality of components to supplement the lists of dimensions.

3. The method of claim 2, wherein exchanging dimensions between the lists in the order indicated by the exchange ordering is repeated until each of the lists includes every dimension included in another one of the lists.

4. The method of claim 1, wherein:

- each exchange ordering indicates an order of a list of the plurality of components; and
- the exchanging dimensions between the lists of dimensions in the order indicated by the exchange ordering to produce the set of supplemented lists of dimensions includes, for each of a plurality of different orderings of a list of the plurality of components:
- for each one of the plurality of components in the order of the list of the plurality of components, supplementing the respective lists of dimensions of other ones of the components with the list of dimensions of the one of the components to produce the supplemented lists of dimensions.

5. The method of claim 4, wherein the supplementing the respective list of dimensions of another one of the components with the list of dimensions of the one of the components includes:

- for each dimension in the list of dimensions of the one of the components, in response to the dimension not being included in the list of dimensions corresponding to the other one of the components, adding the dimension to said list of dimensions corresponding to the other one of the components.

6. The method of claim 5, wherein the adding the dimension to the list of dimensions includes, adding the dimension to the front of the list of dimensions.

7. The method of claim 6, wherein the determining the set of buffers for buffering data between connected ones of the data terminals includes, for each connection between the data terminals, determining a respective additional set of dimensions including a first list of dimensions, corresponding to a first one of the data terminals at a first end of the connection, that appear in a different order than in a second list of dimensions, corresponding to a second one of the data terminals at a second end of the connection.

8. The method of claim 7, wherein the determination of memory requirements for the respective set of buffers includes:

- for each of the respective additional sets of dimensions:
- determining a cardinality of each dimension in the additional set of dimensions; and

14

determining a respective product of the cardinalities of the dimensions in the additional set of dimensions; and

determining a sum of each of the determined products.

9. The method of claim 1, wherein:

- at least one of the plurality of components in the circuit design is described by an IP-Core; and
- the determination of the respective list of dimensions of data used by the data terminal of the at least one of the components includes retrieving meta data from the IP-Core that indicates dimensions of the IP core.

10. The method of claim 1, wherein the plurality of components are connected in series.

11. A method, comprising:

- receiving a specification having a plurality of components, each component having a data terminal connected to a data terminal of another one of the components;
- for each of the data terminals, determining a respective list of dimensions of data used by the data terminal;
- performing on at least one programmed processor, operations including:
- generating a plurality of propagation scenarios, each propagation scenario indicating a set of supplemented lists resulting from a respective exchange of dimensions between the lists of dimensions such that each supplemented list in the set includes every dimension included in the lists of dimensions, the respective exchange of dimensions for each propagation scenario having a different order than the exchanges of scenarios corresponding to other ones of the plurality of propagation scenarios;
- for each propagation scenario, determining a respective set of buffers for buffering data between connected ones of the data terminals based on the corresponding supplemented list of dimensions;
- determining memory requirements for each of the set of buffers corresponding to each exchange ordering; and
- modifying the specification to include the one of the determined sets of buffers having a lowest memory requirement.

12. The method of claim 11, wherein the generation of the plurality of propagation scenarios includes:

- for each of the plurality of propagation scenarios, generating a respective exchange ordering, each of the exchange orderings indicating a different order in which dimensions are exchanged between the lists; and
- for each exchange ordering, exchanging dimensions between the lists of dimensions in the order indicated by the exchange ordering to produce the set of supplemented lists of dimensions of the corresponding propagation scenario.

13. The method of claim 12, wherein:

- each of the exchange orderings indicates an order in which dimensions are exchanged between pairs of connected ones of the plurality of components to produce the corresponding set of supplemented lists of dimensions; and
- the exchanging of dimensions between the lists in the order indicated by the exchange ordering is repeated until each of the lists includes every dimension included in the lists of dimensions.

14. The method of claim 12, wherein:

- each exchange ordering indicates a respective order of a list of the plurality of components;
- the exchanging of the dimensions between the lists of dimensions in the order indicated by the exchange order-

15

ing to produce the set of supplemented lists of dimensions of the corresponding propagation scenario includes:

for each one of the plurality of components in the respective order of the list of the plurality of components, supplementing the respective lists of dimensions of other ones of the components with the list of dimensions of the one of the components to produce the supplemented lists of dimensions; and

the supplementation of the respective list of dimensions of another one of the components with the list of dimensions of the one of the components includes:

for each dimension in the list of dimensions of the one of the components, in response to the dimension not being included in the list of dimensions corresponding to the other one of the components, adding the dimension to said list of dimensions corresponding to the other one of the components.

15. The method of claim **12**, wherein:

the exchanging of dimensions between the lists of dimensions in the order indicated by the exchange ordering to produce the set of supplemented lists adds dimensions to the front of the lists of dimensions according to the exchange ordering; and

the determination of the respective set of buffers for buffering data between connected ones of the data terminals based on the corresponding supplemented list of dimen-

16

sions, of the set of buffers for buffering data between connected ones of the data terminals includes:

for each connection between the data terminals, determining a respective additional set of dimensions including a first list of dimensions, corresponding to a first one of the data terminals at a first end of the connection, that appear in a different order than in a second list of dimensions, corresponding to a second one of the data terminals at a second end of the connection.

16. The method of claim **11**, wherein:

at least one of the plurality of components in the specification is a custom defined component; and further comprising:

in response to determining the set of buffers having the lowest memory requirement, displaying a set of dimensions for the custom designed component based on the set of buffers having the lowest memory requirement.

17. The method of claim **11**, further comprising:

identifying at least one component of the plurality of components in which redundant processing is performed for one or more dimensions included in a buffer preceding the at least one component in the plurality of components; and

modifying the specification to include a circuit after the at least one component to replicate data for the one or more dimensions.

* * * * *